ROS1

ROS1

1.1 ROS1 설치

- ROS 배포판이란?

ROS(로봇 운영 체제) 배포판은 ROS 패키지들의 버전이 지정된 집합으로, 우분투와 같은 리눅스 배포판과 유사합니다. 이러한 배포판은 개발자들이 새로운 버전으로 업데이트할 준비가 될 때까지 안정적인 코드베이스를 사용할 수 있도록 합니다. 배포판이 출시된 후에는 핵심 패키지(`ros-desktop-full`에 포함된 모든 것)에 대해 버그 수정 및 비파괴적 개선만 허용하여 안정성을 유지하려고 합니다. 이는 일반적으로 커뮤니티 전체에 적용되지만, "상위" 수준 패키지의 경우 규칙이 덜 엄격하며, 해당 패키지의 유지 관리자가 파괴적 변화를 피하도록 권장됩니다.

배포판을 구성하는 구성 요소는 **rosdistro** 형식으로 정의되며, 여러 배포판을 지원합니다. 앞으로는 다양한 로봇 플랫폼의 요구 사항에 맞춰 커뮤니티에서 자체 배포판을 만들 가능성이 있습니다.

- ROS 배포판 목록

아래는 ROS 배포판의 전체 목록으로, 출시 날짜, 지원 종료(EOL) 날짜 및 주요 세부 사항을 포함합니다:

배포판	출시 날짜	포스터	EOL 날짜	상태
ROS Noetic Ninjemys	2020년 5월 23일	Noetic Ninjemys	2025년 5월 (Focal EOL)	지원됨 (권장)

ROS Melodic Morenia	2018년 5월 23일	Melodic Morenia	2023년 6월 27일	지원 종료
ROS Lunar Loggerhead	2017년 5월 23일	Lunar Loggerhead	2019년 5월	지원 종료
ROS Kinetic Kame	2016년 5월 23일	Kinetic Kame	2021년 4월 (Xenial EOL)	지원 종료
ROS Jade Turtle	2015년 5월 23일	Jade Turtle	2017년 5월	지원 종료
ROS Indigo Igloo	2014년 7월 22일	I-turtle	2019년 4월 (Trusty EOL)	지원 종료
ROS Hydro Medusa	2013년 9월 4일	H-turtle	2015년 5월	지원 종료
ROS Groovy Galapagos	2012년 12월 31일	G-turtle	2014년 7월	지원 종료
ROS Fuerte Turtle	2012년 4월 23일	F-turtle	--	지원 종료
ROS Electric Emys	2011년 8월 30일	E-turtle	--	지원 종료
ROS Diamondback	2011년 3월 2일	D-turtle	--	지원 종료
ROS C Turtle	2010년 8월 2일	C-turtle	--	지원 종료
ROS Box Turtle	2010년 3월 2일	B-turtle	--	지원 종료

- 배포판 세부 정보

배포판, 지원 플랫폼, 종속성 및 기타 고려 사항에 대한 자세한 정보는 공식 ROS 웹사이트에서 제공되는 **Target Platforms (REP 3)** 문서를 참조하세요.

- 출시 일정 및 정책
- 출시 규칙
  - **ROS 출시 시기**는 필요와 가용 자원에 따라 결정됩니다.
  - 모든 향후 **ROS 1 출시**는 장기 지원(LTS)으로, **5년간 지원**됩니다.
  - ROS 배포판은 지원 종료(EOL)된 우분투 배포판에 대한 지원을 중단하며, 이는 ROS 배포판이 여전히 지원 중이더라도 적용됩니다.
- 출시 정책의 부수적 효과
  - 각 ROS 배포판은 **단일 우분투 LTS 버전**에서만 지원됩니다.

    - LTS 배포판은 이전 배포판과 동일한 우분투 버전을 공유하지 않습니다.
    - ROS 배포판은 출시 이후 새로운 우분투 배포판에 대한 지원을 추가하지 않습니다.

> 참고: 이러한 규칙은 우분투의 출시 정책 변경에 따라 변경될 수 있습니다. 자세한 내용은 ROS 웹사이트의 공식 Release Policy를 참조하세요.

- 예정된 출시
    - **Noetic Ninjemys**는 Open Robotics의 마지막 ROS 1 배포판입니다.
    - 향후 모든 ROS 배포판은 **ROS 2**를 기반으로 하며, 공식 ROS 2 문서 페이지 (`docs.ros.org`)에 나열됩니다.

- 어떤 배포판을 사용해야 하나?

사용 사례에 따라 적합한 ROS 배포판이 달라집니다. 아래는 일반적인 시나리오에 따른 권장 사항입니다 (2020년 5월 기준):

사용 사례	권장 배포판	선호하지만 필수는 아님	비선호
신규 기능	최신 (Noetic)	--	이전 LTS (Melodic)
2년마다 주요 업데이트	이전 LTS (Melodic)	최신 (Noetic)	--
특정 플랫폼 필요	REP-3에서 지원 플랫폼 확인	--	--
최신 Gazebo 필요	Noetic (Gazebo 11)	--	--
OpenCV3 사용	Kinetic, Melodic, Noetic	--	--
OpenCV4 사용	Noetic	--	--

- 지원 플랫폼
- 공식 지원
    - **우분투 (x86_64, armhf)**: ROS 배포판의 주요 플랫폼.
    - **소스 설치**: 모든 배포판에 대해 사용 가능.
- 실험적 플랫폼

이들 플랫폼은 부분적 지원 또는 커뮤니티 기여 설치 가이드를 제공하며, 최신 ROS 배포판과 동기화되지 않거나 일부 패키지만 설치될 수 있습니다:

- 실험적 지원 플랫폼: OS X (Homebrew, MacPorts), Android (NDK), Debian, OpenEmbedded/Yocto, Arch Linux, Windows, Ångström, UDOO, Fedora, Gentoo, OpenSUSE, Raspbian, QNX Realtime OS, Slackware, FreeBSD

- 설치 가이드 (ROS Noetic)
  - *우분투 Focal (20.04 LTS)**에 **ROS Noetic Ninjemys**를 설치하려면 다음 단계를 따르세요.

1단계: 우분투 저장소 구성

우분투 저장소가 **"restricted," "universe," "multiverse"**를 허용하도록 설정하세요. 자세한 방법은 공식 우분투 가이드를 참조하세요.

2단계: `sources.list` 설정

ROS 패키지 저장소를 시스템에 추가:

```
sudo sh -c 'echo "deb <http://packages.ros.org/ros/ubuntu> $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

> 참고: 소스 Debs도 사용 가능합니다. 필요 시 미러를 사용할 수 있습니다.

3단계: 키 설정

`curl`이 설치되지 않았다면 설치하고, ROS 저장소 키를 추가:

```
sudo apt install curl
curl -s <https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc> | sudo apt-key add -
```

4단계: 패키지 인덱스 업데이트

Debian 패키지 인덱스를 업데이트:

```
sudo apt update
```

5단계: ROS 설치

원하는 설치 수준을 선택:

- **Desktop-Full 설치** (권장): Desktop에 포함된 모든 것과 2D/3D 시뮬레이터 및 인식 패키지 포함.

  ```
  sudo apt install ros-noetic-desktop-full
  ```

- **Desktop 설치**: ROS-Base에 `rqt`, `rviz`와 같은 도구 포함.

  ```
  sudo apt install ros-noetic-desktop
  ```

- **ROS-Base**: 패키징, 빌드, 통신 라이브러리만 포함 (GUI 도구 제외).

  ```
  sudo apt install ros-noetic-ros-base
  ```

특정 패키지 설치:

```
sudo apt install ros-noetic-PACKAGE
```

예시:

```
sudo apt install ros-noetic-slam-gmapping
```

사용 가능한 패키지 검색:

```
apt search ros-noetic
```

6단계: 환경 설정

ROS를 사용할 모든 터미널에서 ROS 설정 스크립트를 실행:

```
source /opt/ros/noetic/setup.bash
```

새 터미널마다 자동으로 실행되도록 설정:

- **Bash**:

  ```
  echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
  source ~/.bashrc
  ```

- **Zsh**:

```
echo "source /opt/ros/noetic/setup.zsh" >> ~/.zshrc
source ~/.zshrc
```

> 참고: 여러 ROS 배포판이 설치된 경우, ~/.bashrc 또는 ~/.zshrc가 원하는 버전만 실행하도록 설정하세요.

7단계: 패키지 빌드 종속성 설치

ROS 작업 공간을 생성하고 관리하려면 추가 도구를 설치:

```
sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-gene
rator python3-wstool build-essential
```

8단계: `rosdep` 초기화

시스템 종속성 관리를 위해 `rosdep`을 설치 및 초기화:

```
sudo apt install python3-rosdep
sudo rosdep init
rosdep update
```

서드파티 설치 대안

- **Nootrix Built VM**: 우분투 14.04.1 LTS와 ROS Indigo가 사전 설치된 가상 머신으로, VirtualBox 또는 기타 가상화 엔진에서 실행 가능한 `.ova` 파일로 제공.
- **robotpkg**: *NIX 및 BSD 시스템용 소스 기반 패키지 관리자.
- **TwoLineInstall**: 우분투 13.10 또는 14.04 LTS에 ROS Indigo를 설치하기 위한 간단한 스크립트.
- **OS X 설치 스크립트**: OS X Yosemite 또는 El Capitan에서 ROS를 설치하기 위한 스크립트 기반 설치 절차.

1.2 Docker 를 이용한 ROS1 설치

1.3 패키지 생성

- 환경 설정

ROS 설치를 진행할 때, 여러 `setup.*sh` 파일을 소싱(source)하라는 안내가 나타납니다. 또한, 이 소싱 명령을 셸 시작 스크립트(예: `.bashrc`)에 추가할 수도 있습니다. 이는 ROS가 셸 환경을 활용하여 작업 공간을 결합하는 개념에 의존하기 때문에 필요합니다. 이를 통해 서로 다른 ROS 버전이나 패키지 집합을 쉽게 개발할 수 있습니다.

만약 ROS 패키지를 찾거나 사용하는 데 문제가 있다면, 환경이 올바르게 설정되었는지 확인하세요. 환경 변수 `ROS_ROOT`와 `ROS_PACKAGE_PATH`가 설정되어 있는지 확인하는 것이 좋은 방법입니다:

```
$ printenv | grep ROS
```

이 변수들이 설정되어 있지 않다면, `setup.*sh` 파일을 소싱해야 할 수 있습니다.

`setup.*sh` 파일의 출처

환경 설정 파일은 다양한 곳에서 생성됩니다:

- 패키지 매니저로 설치된 ROS 패키지는 `setup.*sh` 파일을 제공합니다.
- `rosws`와 같은 도구를 사용해 생성된 rosbuild 작업 공간은 `setup.*sh` 파일을 제공합니다.
- catkin 패키지를 빌드하거나 설치할 때 `setup.*sh` 파일이 부산물로 생성됩니다.

> 참고: 튜토리얼 전반에 걸쳐 rosbuild와 catkin에 대한 언급이 등장합니다. 이는 ROS 코드를 구성하고 빌드하는 두 가지 방법입니다. rosbuild는 더 이상 권장되지 않으며 레거시로 유지됩니다. catkin은 표준 CMake 규칙을 따르며, 외부 코드베이스 통합이나 소프트웨어 배포를 원하는 사용자에게 더 유연성을 제공하므로 권장됩니다. 자세한 내용은 catkin 문서 또는 rosbuild 문서를 참조하세요.

우분투에서 `apt`를 통해 ROS를 설치했다면, `/opt/ros/<배포판 이름>/` 경로에 `setup.*sh` 파일이 있습니다. 예를 들어, 다음과 같이 소싱할 수 있습니다:

```
$ source /opt/ros/<배포판 이름>/setup.bash
```

ROS Kinetic을 설치했다면, 다음과 같이 실행:

```
$ source /opt/ros/kinetic/setup.bash
```

ROS 명령어에 접근하려면 새 셸을 열 때마다 이 명령어를 실행해야 합니다. 또는 `.bashrc`에 이 명령어를 추가하여 자동화할 수 있습니다. 이를 통해 동일한 컴퓨터에 여러 ROS 배포판(예: Indigo, Kinetic)을 설치하고 전환할 수 있습니다.

다른 플랫폼에서는 ROS를 설치한 위치에서 `setup.*sh` 파일을 찾을 수 있습니다.

- ROS 작업 공간 생성

> 참고: 이 지침은 ROS Groovy 이상에 적용됩니다. ROS Fuerte 이하에서는 rosbuild를 선택하세요.

catkin 작업 공간을 생성하고 빌드해 봅시다:

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

`catkin_make`는 catkin 작업 공간을 다루기 위한 편리한 도구입니다. 작업 공간에서 처음 실행하면 `src` 폴더에 `CMakeLists.txt` 링크를 생성합니다.

> Python 3 사용자 주의 (ROS Melodic 이하): Python 3 호환성을 위해 소스에서 ROS를 빌드하고, 필요한 ROS Python 패키지의 Python 3 버전(예: catkin)을 설치한 경우, 깨끗한 catkin 작업 공간에서 첫 번째 catkin_make 명령은 다음과 같이 실행해야 합니다:

```
$ catkin_make -DPYTHON_EXECUTABLE=/usr/bin/python3
```

이렇게 하면 `catkin_make`가 Python 3으로 구성됩니다. 이후 빌드에서는 `catkin_make`만 사용하면 됩니다.

현재 디렉토리를 살펴보면 `build`와 `devel` 폴더가 생성된 것을 확인할 수 있습니다. `devel` 폴더 안에는 여러 `setup.*sh` 파일이 있습니다. 이 파일을 소싱하면 현재 작업 공간이 환경에 오버레이됩니다. 자세한 내용은 catkin 문서를 참조하세요. 계속 진행하기 전에 새로 생성된 `setup.*sh` 파일을 소싱하세요:

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

작업 공간이 올바르게 오버레이되었는지 확인하려면 `ROS_PACKAGE_PATH` 환경 변수에 현재 디렉토리가 포함되어 있는지 확인하세요:

```
$ echo $ROS_PACKAGE_PATH
```

출력 예시:

```
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

이제 환경 설정이 완료되었으므로, ROS 파일 시스템 튜토리얼로 넘어갈 수 있습니다.

- 파일 시스템 개념 개요
  - **패키지(Packages)**: ROS 코드의 소프트웨어 조직 단위로, 라이브러리, 실행 파일, 스크립트 또는 기타 아티팩트를 포함할 수 있습니다.
  - **매니페스트(package.xml)**: 패키지에 대한 설명으로, 패키지 간 종속성을 정의하고 버전, 유지 관리자, 라이선스 등의 메타 정보를 기록합니다.

- 파일 시스템 도구

ROS 코드는 많은 패키지에 걸쳐 분산되어 있습니다. `ls`나 `cd` 같은 명령줄 도구로 탐색하는 것은 번거로울 수 있으므로, ROS는 이를 돕기 위한 도구를 제공합니다.

`rospack` 사용

`rospack`은 패키지에 대한 정보를 얻을 수 있는 도구입니다. 이 튜토리얼에서는 `find` 옵션만 다룹니다. 이 옵션은 패키지의 경로를 반환합니다.

사용법:

```
$ rospack find [패키지_이름]
```

예시:

```
$ rospack find roscpp
```

출력:

```
YOUR_INSTALL_PATH/share/roscpp
```

우분투에서 `apt`로 ROS Kinetic을 설치했다면 정확히 다음과 같이 표시됩니다:

```
/opt/ros/kinetic/share/roscpp
```

`roscd` 사용

`roscd`는 `rosbash` 스위트의 일부로, 패키지나 스택으로 바로 디렉토리를 변경할 수 있습니다.

사용법:

```
$ roscd <패키지-또는-스택>[/하위디렉토리]
```

`roscpp` 패키지 디렉토리로 이동했는지 확인하려면:

```
$ roscd roscpp
$ pwd
```

출력:

```
YOUR_INSTALL_PATH/share/roscpp
```

이는 앞서 `rospack find`로 얻은 경로와 동일합니다.

> 참고: roscd와 같은 ROS 도구는 ROS_PACKAGE_PATH에 나열된 디렉토리 내의 ROS 패키지만 찾습니다. ROS_PACKAGE_PATH를 확인하려면:

```
$ echo $ROS_PACKAGE_PATH
```

`ROS_PACKAGE_PATH`에는 ROS 패키지가 있는 디렉토리 목록이 콜론(`:`)으로 구분되어 표시됩니다. 예시:

```
/opt/ros/kinetic/base/install/share
```

다른 환경 경로와 마찬가지로, `ROS_PACKAGE_PATH`에 추가 디렉토리를 콜론으로 구분하여 추가할 수 있습니다.

- 하위 디렉토리

`roscd`는 패키지나 스택의 하위 디렉토리로도 이동할 수 있습니다. 예:

```
$ roscd roscpp/cmake
$ pwd
```

출력:

```
YOUR_INSTALL_PATH/share/roscpp/cmake
```

`roscd log`

`roscd log`는 ROS가 로그 파일을 저장하는 폴더로 이동합니다. 아직 ROS 프로그램을 실행하지 않았다면, 이 디렉토리가 존재하지 않는다는 에러가 발생할 수 있습니다.

ROS 프로그램을 실행한 적이 있다면:

```
$ roscd log
```

`rosls` 사용

`rosls`는 `rosbash` 스위트의 일부로, 절대 경로 대신 패키지 이름으로 직접 `ls` 명령을 실행할 수 있습니다.

사용법:

```
$ rosls <패키지-또는-스택>[/하위디렉토리]
```

예시:

```
$ rosls roscpp_tutorials
```

출력:

```
cmake launch package.xml srv
```

- 탭 완성

패키지 이름을 전부 입력하는 것은 번거로울 수 있습니다. 예를 들어, `roscpp_tutorials`는 꽤 긴 이름입니다. 다행히 일부 ROS 도구는 탭 완성을 지원합니다.

예시:

```
$ roscd roscpp_tut
```

이제 **탭 키**를 누르면:

```
$ roscd roscpp_tutorials/
```

탭 완성은 `roscpp_tut`로 시작하는 유일한 패키지가 `roscpp_tutorials`이기 때문에 작동합니다.

다른 예시:

```
$ roscd tur
```

탭 키를 누르면:

```
$ roscd turtle
```

`turtle`로 시작하는 패키지가 여러 개일 경우, 한 번 더 탭 키를 누르면 목록이 표시됩니다:

```
turtle_actionlib/  turtlesim/       turtle_tf/
```

명령줄에는 여전히:

```
$ roscd turtle
```

이제 `turtle` 뒤에 `s`를 입력하고 탭 키를 누르면:

```
$ roscd turtles
```

`turtles`로 시작하는 유일한 패키지가 `turtlesim`이므로:

```
$ roscd turtlesim/
```

모든 설치된 패키지 목록을 보려면:

```
$ rosls
```

그리고 **탭 키**를 두 번 누르세요.

- catkin 패키지 구성 요소

catkin 패키지로 간주되려면 다음 요구 사항을 충족해야 합니다:

- 패키지에 catkin 호환 `package.xml` 파일이 포함되어야 합니다.
  - 이 파일은 패키지에 대한 메타 정보를 제공합니다.
- 패키지에 catkin을 사용하는 `CMakeLists.txt` 파일이 있어야 합니다.
- catkin 메타패키지인 경우, 관련 표준 `CMakeLists.txt` 파일이 있어야 합니다.
- 각 패키지는 고유한 폴더를 가져야 하며, 중첩된 패키지나 동일한 디렉토리를 공유하는 여러 패키지는 허용되지 않습니다.

가장 간단한 패키지 구조는 다음과 같습니다:

```
my_package/
  CMakeLists.txt
  package.xml
```

- catkin 작업 공간의 패키지

catkin 패키지를 다루는 권장 방법은 catkin 작업 공간을 사용하는 것이지만, 독립적으로 빌드할 수도 있습니다. 간단한 작업 공간 구조는 다음과 같습니다:

```
workspace_folder/        -- 작업 공간
  src/                   -- 소스 공간
    CMakeLists.txt       -- catkin이 제공하는 최상위 CMake 파일
    package_1/
      CMakeLists.txt     -- package_1의 CMakeLists.txt 파일
      package.xml        -- package_1의 패키지 매니페스트
    ...
    package_n/
      CMakeLists.txt     -- package_n의 CMakeLists.txt 파일
      package.xml        -- package_n의 패키지 매니페스트
```

이 튜토리얼을 계속 진행하기 전에, catkin 작업 공간 생성 튜토리얼을 따라 빈 catkin 작업 공간을 생성하세요.

- catkin 패키지 생성

이 튜토리얼에서는 `catkin_create_pkg` 스크립트를 사용해 새로운 catkin 패키지를 생성하는 방법과 생성 후 할 수 있는 작업을 알아봅니다.

먼저, catkin 작업 공간 생성 튜토리얼에서 만든 catkin 작업 공간의 소스 공간 디렉토리로 이동:

```
$ cd ~/catkin_ws/src
```

이제 `catkin_create_pkg` 스크립트를 사용해 `beginner_tutorials`라는 패키지를 생성하고, `std_msgs`, `roscpp`, `rospy`에 의존하도록 설정:

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

이렇게 하면 `beginner_tutorials` 폴더가 생성되고, `catkin_create_pkg`에 제공한 정보로 부분적으로 채워진 `package.xml`과 `CMakeLists.txt` 파일이 포함됩니다.

`catkin_create_pkg`는 패키지 이름과 선택적으로 의존하는 패키지 목록을 필요로 합니다:

```
# 예시, 실행하지 마세요
$ catkin_create_pkg <패키지_이름> [의존성1] [의존성2] [의존성3]
```

`catkin_create_pkg`의 고급 기능은 catkin 명령 문서에서 확인할 수 있습니다.

- catkin 작업 공간 빌드 및 설정 파일 소싱

이제 catkin 작업 공간의 패키지를 빌드해야 합니다:

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

작업 공간이 빌드되면, `/opt/ros/$ROSDISTRO_NAME`에서와 유사한 구조가 `devel` 하위 폴더에 생성됩니다.

작업 공간을 ROS 환경에 추가하려면 생성된 설정 파일을 소싱:

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

- 패키지 종속성

1차 종속성

앞서 `catkin_create_pkg`를 사용할 때 몇 가지 패키지 종속성을 지정했습니다. 이제 `rospack` 도구로 이 1차 종속성을 확인할 수 있습니다:

```
$ rospack depends1 beginner_tutorials
```

출력:

```
roscpp
rospy
std_msgs
```

`rospack`은 `catkin_create_pkg` 실행 시 인수로 사용된 동일한 종속성을 나열합니다. 패키지의 종속성은 `package.xml` 파일에 저장됩니다:

```
$ roscd beginner_tutorials
$ cat package.xml
```

```
<package format="2">
...
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
...
</package>
```

- 간접 종속성

종속성은 자체적으로 다른 종속성을 가질 수 있습니다. 예를 들어, `rospy`는 다른 종속성을 가집니다:

```
$ rospack depends1 rospy
```

출력:

```
genpy
roscpp
rosgraph
rosgraph_msgs
```

```
roslib
std_msgs
```

패키지는 많은 간접 종속성을 가질 수 있습니다. 다행히 `rospack`은 재귀적으로 모든 중첩된 종속성을 확인할 수 있습니다:

```
$ rospack depends beginner_tutorials
```

출력 예시:

```
cpp_common
rostime
roscpp_traits
roscpp_serialization
catkin
genmsg
genpy
message_runtime
gencpp
geneus
gennodejs
genlisp
message_generation
rosbuild
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
ros_environment
rospack
roslib
rospy
```

- 패키지 커스터마이징

이제 `catkin_create_pkg`로 생성된 파일을 하나씩 살펴보고, 각 구성 요소를 줄 단위로 설명하며 패키지에 맞게 커스터마이징하는 방법을 알아봅니다.

`package.xml` 커스터마이징

생성된 `package.xml`은 새 패키지에 포함되어 있습니다. 이제 이 파일을 살펴보고 수정이 필요한 부분을 조정해 봅시다.

`<description>` 태그

먼저 `<description>` 태그를 업데이트:

```
<description>The beginner_tutorials package</description>
```

설명을 원하는 대로 변경하세요. 관례상 첫 문장은 패키지의 범위를 다루며 짧아야 합니다. 한 문장으로 설명하기 어렵다면 패키지를 분리해야 할 수 있습니다.

`<maintainer>` 태그

다음은 `<maintainer>` 태그입니다:

```
<!-- One maintainer tag required, multiple allowed, one person per tag -->
<!-- Example:  -->
<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer>
-->
<maintainer email="user@todo.todo">user</maintainer>
```

이 태그는 패키지에 대한 연락처를 알리는 필수 태그입니다. 최소 한 명의 유지 관리자가 필요하며, 여러 명을 지정할 수도 있습니다. 유지 관리자의 이름은 태그 본문에, 이메일은 속성으로 입력:

```
<maintainer email="you@yourdomain.tld">Your Name</maintainer>
```

`<license>` 태그

다음은 필수인 `<license>` 태그:

```
<!-- One license tag required, multiple allowed, one license per tag -->
<!-- Commonly used license strings: -->
<!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3
-->
<license>TODO</license>
```

라이선스를 선택해 입력하세요. 일반적인 오픈소스 라이선스는 BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 등이 있습니다. 이 튜토리얼에서는 ROS 핵심 구성 요소가 사용하는 BSD를 사용:

```
<license>BSD</license>
```

- 종속성 태그

다음 태그는 패키지의 종속성을 설명합니다. 종속성은 `build_depend`, `buildtool_depend`, `exec_depend`, `test_depend`로 나뉩니다. 자세한 설명은 Catkin 종속성 문서를 참조하세요. `catkin_create_pkg`에 `std_msgs`, `roscpp`, `rospy`를 인수로 전달했으므로 종속성은 다음과 같이 표시:

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>genmsg</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use exec_depend for packages you need at runtime: -->
<!--   <exec_depend>python-yaml</exec_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
```

지정한 모든 종속성은 기본 `buildtool_depend`(catkin)와 함께 `build_depend`로 추가되었습니다. 이 경우 모든 종속성이 빌드와 런타임 모두에 필요하므로, 각 종속성에 대해 `exec_depend` 태그를 추가:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<exec_depend>roscpp</exec_depend>
```

```
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
```

- 최종 `package.xml`

주석과 사용하지 않는 태그를 제거한 최종 `package.xml`은 훨씬 간결합니다:

```
<?xml version="1.0"?>
<package format="2">
  <name>beginner_tutorials</name>
  <version>0.1.0</version>
  <description>The beginner_tutorials package</description>
  <maintainer email="you@yourdomain.tld">Your Name</maintainer>
  <license>BSD</license>
  <url type="website"><http://wiki.ros.org/beginner_tutorials></url>
  <author email="you@yourdomain.tld">Jane Doe</author>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
</package>
```

`CMakeLists.txt` 커스터마이징

`package.xml`을 패키지에 맞게 조정한 후, 메타 정보를 포함하는 작업은 완료되었습니다. `catkin_create_pkg`로 생성된 `CMakeLists.txt` 파일은 이후 ROS 코드 빌드 튜토리얼에서 다룰 예정입니다.

1.4 파이썬 프로그램밍 hello_ros 노드 만들기

- 패키지 빌드

패키지의 모든 시스템 종속성이 설치되어 있다면, 이제 새로 만든 패키지를 빌드할 수 있습니다.

> 참고: apt 또는 다른 패키지 매니저를 통해 ROS를 설치했다면, 모든 종속성이 이미 설치되어 있을 것입니다.

계속 진행하기 전에, 아직 환경 설정 파일을 소싱하지 않았다면 소싱하세요. 우분투에서는 다음과 같이 실행:

```
# source /opt/ros/<ROS_배포판_이름>/setup.bash
$ source /opt/ros/kinetic/setup.bash  # 예: Kinetic
```

`catkin_make` 사용

`catkin_make`는 표준 catkin 워크플로우에 편의를 더해주는 명령줄 도구입니다. `catkin_make`는 표준 CMake 워크플로우에서 `cmake`와 `make` 호출을 결합한 것이라고 생각할 수 있습니다.

- 사용법

```
# catkin 작업 공간에서
$ catkin_make [make_대상] [-DCMAKE_변수=...]
```

표준 CMake 워크플로우에 익숙하지 않은 분들을 위해 간단히 설명하면 다음과 같습니다:

> 참고: 아래 명령은 예시일 뿐이며 실제로 실행하면 작동하지 않습니다.

```
# CMake 프로젝트에서
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install  # (선택 사항)
```

이 과정은 각 CMake 프로젝트마다 실행됩니다. 반면, catkin 프로젝트는 작업 공간에서 함께 빌드할 수 있습니다. 작업 공간에서 0개 이상의 catkin 패키지를 빌드하는 워크플로우는 다음과 같습니다:

```
# catkin 작업 공간에서
$ catkin_make
$ catkin_make install  # (선택 사항)
```

위 명령은 `src` 폴더에 있는 모든 catkin 프로젝트를 빌드합니다. 이는 REP128의 권장 사항을 따릅니다. 소스 코드가 다른 위치(예: `my_src`)에 있다면, 다음과 같이 `catkin_make`를 호출:

> 참고: 아래 명령은 my_src 디렉토리가 존재하지 않으므로 실행하면 작동하지 않습니다.

```
# catkin 작업 공간에서
$ catkin_make --source my_src
$ catkin_make install --source my_src  # (선택 사항)
```

`catkin_make`의 고급 사용법은 catkin 명령 문서를 참조하세요.

- 패키지 빌드

이 페이지를 사용해 자신의 코드를 빌드하려는 경우, 이후 튜토리얼(C++ 또는 Python)을 참고하여 `CMakeLists.txt` 파일을 수정해야 할 수도 있습니다.

이전 튜토리얼(패키지 생성)에서 이미 catkin 작업 공간과 `beginner_tutorials`라는 새 패키지를 만들었을 것입니다. catkin 작업 공간으로 이동하여 `src` 폴더를 확인:

```
$ cd ~/catkin_ws/
$ ls src
```

출력:

```
beginner_tutorials/  CMakeLists.txt@
```

이전 튜토리얼에서 `catkin_create_pkg`로 생성한 `beginner_tutorials` 폴더가 보일 것입니다. 이제 `catkin_make`를 사용해 이 패키지를 빌드:

```
$ catkin_make
```

`cmake`와 `make`로부터 많은 출력이 표시될 것이며, 다음과 비슷할 것입니다:

```
Base path: /home/user/catkin_ws
Source space: /home/user/catkin_ws/src
Build space: /home/user/catkin_ws/build
Devel space: /home/user/catkin_ws/devel
Install space: /home/user/catkin_ws/install
```

```
####
#### Running command: "cmake /home/user/catkin_ws/src
-DCATKIN_DEVEL_PREFIX=/home/user/catkin_ws/devel
-DCMAKE_INSTALL_PREFIX=/home/user/catkin_ws/install" in "/home/user/
catkin_ws/build"
####
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is Clang 4.0.0
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - ye
s
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Using CATKIN_DEVEL_PREFIX: /tmp/catkin_ws/devel
-- Using CMAKE_PREFIX_PATH: /opt/ros/kinetic
-- This workspace overlays: /opt/ros/kinetic
-- Found PythonInterp: /usr/bin/python (found version "2.7.1")
-- Found PY_em: /usr/lib/python2.7/dist-packages/em.pyc
-- Found gtest: gtests will be built
-- catkin 0.5.51
-- BUILD_SHARED_LIBS is on
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing packages in topological order:
-- ~~  - beginner_tutorials
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ add_subdirectory(beginner_tutorials)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/catkin_ws/build
####
```

```
#### Running command: "make -j4" in "/home/user/catkin_ws/build"
####
```

`catkin_make`는 먼저 각 '공간(space)'에 사용된 경로를 표시합니다. 이러한 공간은 REP128과 catkin 작업 공간 문서에 설명되어 있습니다. 중요한 점은 이러한 기본값으로 인해 catkin 작업 공간에 여러 폴더가 생성되었다는 것입니다. 확인해 보세요:

```
$ ls
```

출력:

```
build  devel  src
```

- **build 폴더**: 빌드 공간의 기본 위치로, `cmake`와 `make`가 호출되어 패키지를 구성하고 빌드합니다.
- **devel 폴더**: 개발 공간의 기본 위치로, 패키지를 설치하기 전에 실행 파일과 라이브러리가 저장됩니다.

- ROS 그래프 개념 개요

이제 패키지를 빌드했으니, ROS 노드와 관련된 개념을 살펴보겠습니다.

- **노드(Nodes)**: 다른 노드와 통신하기 위해 ROS를 사용하는 실행 파일입니다.
- **메시지(Messages)**: 토픽에 구독하거나 발행할 때 사용되는 ROS 데이터 유형입니다.
- **토픽(Topics)**: 노드는 토픽에 메시지를 발행하거나 토픽을 구독하여 메시지를 수신할 수 있습니다.
- **마스터(Master)**: ROS의 이름 서비스로, 노드들이 서로를 찾을 수 있도록 도와줍니다.
- **rosout**: ROS의 `stdout/stderr`에 해당합니다.
- **roscore**: 마스터, rosout, 파라미터 서버(나중에 소개)를 포함합니다.

- 노드

노드는 ROS 패키지 내의 실행 파일일 뿐입니다. ROS 노드는 ROS 클라이언트 라이브러리를 사용해 다른 노드와 통신합니다. 노드는 토픽에 발행하거나 구독할 수 있으며, 서비스를 제공하거나 사용할 수도 있습니다.

- 클라이언트 라이브러리

ROS 클라이언트 라이브러리는 서로 다른 프로그래밍 언어로 작성된 노드 간 통신을 가능하게 합니다:

- `rospy`: Python 클라이언트 라이브러리
- `roscpp`: C++ 클라이언트 라이브러리

`roscore`

`roscore`는 ROS를 사용할 때 가장 먼저 실행해야 하는 프로그램입니다.

다음 명령을 실행:

```
$ roscore
```

다음과 비슷한 출력이 표시됩니다:

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/roslaunch
-machine_name-13039.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://machine_name:33919/
ros_comm version 1.4.7

SUMMARY
======

PARAMETERS
 * /rosversion
 * /rosdistro

NODES

auto-starting new master
process[master]: started with pid [13054]
ROS_MASTER_URI=http://machine_name:11311/

setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
```

```
process[rosout-1]: started with pid [13067]
started core service [/rosout]
```

`roscore`가 초기화되지 않는다면, 네트워크 설정에 문제가 있을 수 있습니다. 네트워크 설정 - 단일 머신 구성을 참조하세요.

`roscore`가 권한 부족 메시지와 함께 초기화되지 않는다면, `~/.ros` 폴더의 소유자가 `root`일 가능성이 있습니다. 다음 명령으로 소유권을 재귀적으로 변경:

```
$ sudo chown -R <사용자_이름> ~/.ros
```

`rosnode` 사용

새 터미널을 열고, `rosnode`를 사용해 `roscore` 실행 결과를 확인해 보겠습니다. 이전 터미널은 열어 둔 상태로 유지하세요(새 탭을 열거나 최소화).

> 참고: 새 터미널을 열면 환경이 초기화되고 ~/.bashrc 파일이 소싱됩니다. rosnode 같은 명령어가 작동하지 않는다면, ~/.bashrc에 환경 설정 파일을 추가하거나 수동으로 재소싱해야 할 수 있습니다.

`rosnode`는 현재 실행 중인 ROS 노드에 대한 정보를 표시합니다. `rosnode list` 명령으로 활성 노드를 나열:

```
$ rosnode list
```

출력:

```
/rosout
```

현재 실행 중인 노드는 `rosout`뿐입니다. 이 노드는 항상 실행되며, 노드의 디버깅 출력을 수집하고 기록합니다.

`rosnode info` 명령은 특정 노드에 대한 자세한 정보를 반환:

```
$ rosnode info /rosout
```

출력 예시:

```
--------------------------------------------------------------------------------
Node [/rosout]
```

```
Publications:
 * /rosout_agg [rosgraph_msgs/Log]

Subscriptions:
 * /rosout [unknown type]

Services:
 * /rosout/get_loggers
 * /rosout/set_logger_level

contacting node http://machine_name:54614/ ...
Pid: 5092
```

`rosout`이 `/rosout_agg`를 발행하는 등 추가 정보를 확인할 수 있습니다.

이제 더 많은 노드를 확인해 보겠습니다. 이를 위해 `rosrun`을 사용해 다른 노드를 실행합니다.

- `rosrun` 사용

`rosrun`은 패키지 경로를 알지 않아도 패키지 이름으로 노드를 직접 실행할 수 있게 해줍니다.

사용법

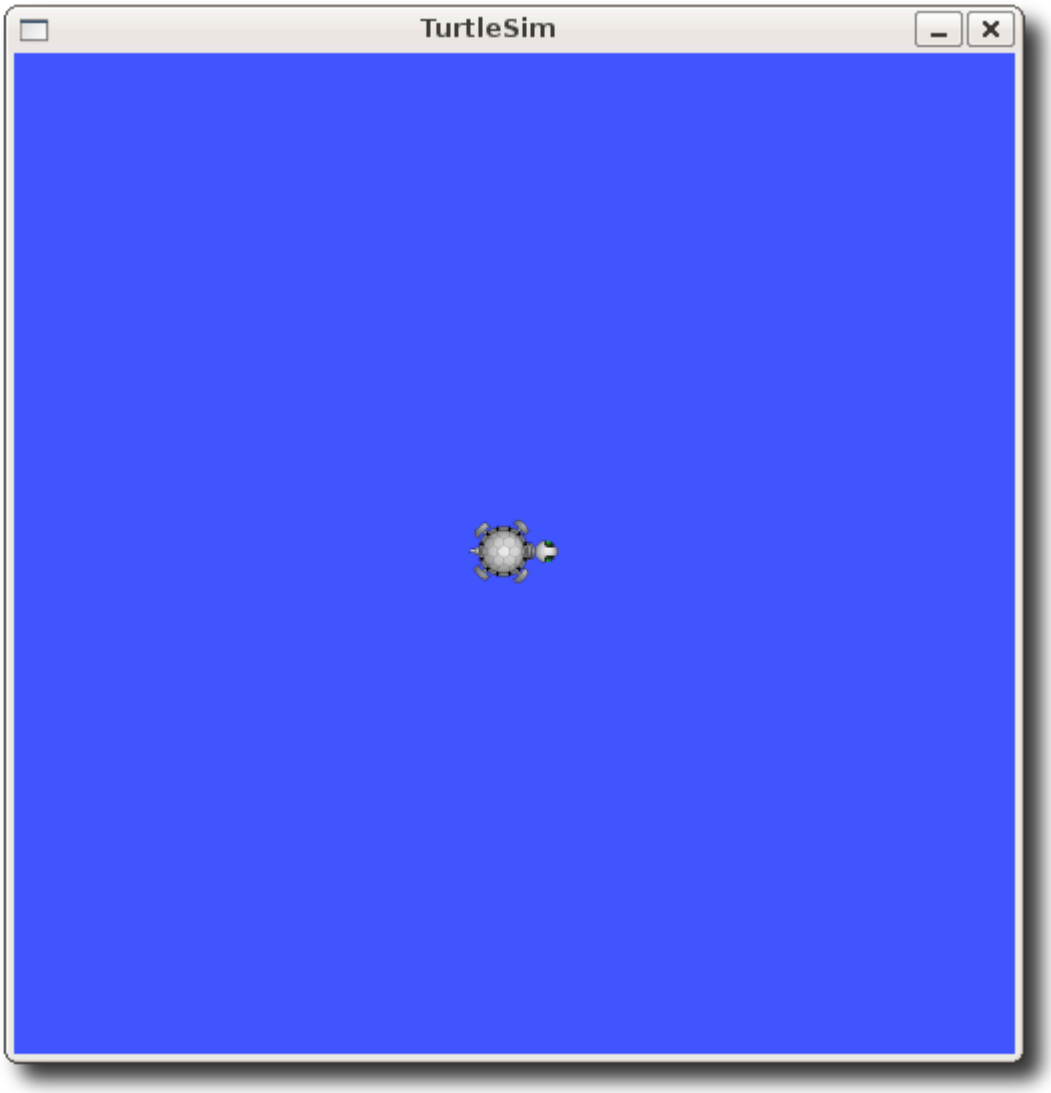
```
$ rosrun [패키지_이름] [노드_이름]
```

이제 `turtlesim` 패키지의 `turtlesim_node`를 실행:

새 터미널에서:

```
$ rosrun turtlesim turtlesim_node
```

`turtlesim` 창이 나타납니다:



> 참고: turtlesim 창의 거북이 모양은 다를 수 있습니다. 다양한 거북이 유형이 있으니 놀라지 마세요!

새 터미널에서 노드 목록을 확인:

```
$ rosnode list
```

출력:

```
/rosout
/turtlesim
```

ROS의 강력한 기능 중 하나는 명령줄에서 노드 이름을 재지정할 수 있다는 점입니다.

`turtlesim` 창을 닫아 노드를 중지하거나, `rosrun turtlesim` 터미널로 돌아가 `Ctrl-C`를 사용하세요. 이제 노드 이름을 변경하는 리매핑 인수를 사용해 다시 실행:

```
$ rosrun turtlesim turtlesim_node __name:=my_turtle
```

다시 노드 목록을 확인:

```
$ rosnode list
```

출력:

```
/my_turtle
/rosout
```

> 참고: /turtlesim이 여전히 목록에 있다면, 창을 닫지 않고 터미널에서 Ctrl-C로 노드를 중지했거나, 네트워크 설정 - 단일 머신 구성에 설명된 $ROS_HOSTNAME 환경 변수가 정의되지 않았을 수 있습니다. 다음 명령으로 노드 목록을 정리:

```
$ rosnode cleanup
```

새로운 `/my_turtle` 노드가 표시됩니다. `rosnode ping` 명령으로 노드가 실행 중인지 테스트:

```
$ rosnode ping my_turtle
```

출력:

```
rosnode: node is [/my_turtle]
pinging /my_turtle with a timeout of 3.0s
xmlrpc reply from <http://aqy:42235/>     time=1.152992ms
xmlrpc reply from <http://aqy:42235/>     time=1.120090ms
xmlrpc reply from <http://aqy:42235/>     time=1.700878ms
xmlrpc reply from <http://aqy:42235/>     time=1.127958ms
```

- 검토

이번 섹션에서 다룬 내용:

- **roscore** : ros+core로, 마스터(ROS 이름 서비스), rosout(stdout/stderr), 파라미터 서버(나중에 소개)를 포함.
- **rosnode** : ros+node로, 노드 정보를 얻는 ROS 도구.
- **rosrun** : ros+run으로, 주어진 패키지의 노드를 실행.

- 퍼블리셔 노드 작성

ROS에서 "노드(Node)"는 ROS 네트워크에 연결된 실행 파일을 의미합니다. 여기서는 지속적으로 메시지를 브로드캐스트하는 퍼블리셔 노드("talker")를 만들어 보겠습니다.

이전 튜토리얼(패키지 생성)에서 만든 beginner_tutorials 패키지 디렉토리로 이동:

```
$ roscd beginner_tutorials
```

- 코드 작성

먼저 Python 스크립트를 저장할 scripts 폴더를 생성:

```
$ mkdir scripts
$ cd scripts
```

예제 스크립트 talker.py를 다운로드하고 실행 가능하도록 설정:

```
$ wget <https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/talker.py>
$ chmod +x talker.py
```

> 참고: 아직 실행하지 마세요. 파일을 확인하거나 수정하려면 다음 명령을 사용하거나 아래 코드를 참고:

```
$ rosed beginner_tutorials talker.py
```

- talker.py 코드

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

- `CMakeLists.txt`에 다음을 추가하여 Python 스크립트가 올바르게 설치되고 적절한 Python 인터프리터를 사용하도록 설정:

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

- 코드 설명

`talker.py` 코드를 줄 단위로 분석해 보겠습니다.

1. Python 선언

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

모든 Python ROS 노드는 파일 상단에 이 선언을 포함합니다. 이 줄은 스크립트가 Python 스크립트로 실행되도록 보장합니다.

2. 모듈 임포트

```
import rospy
from std_msgs.msg import String
```

- `rospy`: ROS 노드를 작성하기 위해 필요합니다.
- `std_msgs.msg.String`: 퍼블리싱에 사용할 간단한 문자열 메시지 타입을 재사용하기 위해 임포트합니다.

3. 퍼블리셔 설정

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

- `rospy.Publisher('chatter', String, queue_size=10)`: 노드가 `chatter` 토픽에 `String` 타입 메시지를 퍼블리싱한다고 선언합니다. 여기서 `String`은 `std_msgs.msg.String` 클래스입니다. `queue_size`는 ROS Hydro부터 추가된 인수로, 구독자가 메시지를 충분히 빠르게 수신하지 못할 경우 큐에 저장되는 메시지 수를 제한합니다(이전 버전에서는 이 인수를 생략).
- `rospy.init_node('talker', anonymous=True)`: 노드의 이름을 `talker`로 지정하며, ROS 마스터와 통신을 시작하기 위해 필수적입니다. 노드 이름은 슬래시(`/`)를 포함하지 않는 기본 이름이어야 합니다. `anonymous=True`는 노드 이름 끝에 임의의 숫자를 추가하여 고유성을 보장합니다.

4. 루프 속도 설정

```
rate = rospy.Rate(10) # 10hz
```

`Rate` 객체를 생성하여 루프가 원하는 속도(초당 10회)로 실행되도록 합니다. `rate.sleep()` 메서드는 루프 속도를 유지하기 위해 필요한 시간만큼 대기합니다.

5. 메인 루프

```
while not rospy.is_shutdown():
    hello_str = "hello world %s" % rospy.get_time()
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
```

- `rospy.is_shutdown()`: 프로그램 종료 여부(예: `Ctrl-C` 입력)를 확인합니다.

- `hello_str`: 현재 시간을 포함한 문자열 생성.
- `rospy.loginfo(hello_str)`: 메시지를 화면에 출력하고, 노드 로그 파일 및 `rosout`에 기록합니다. `rosout`은 디버깅에 유용하며, `rqt_console`로 메시지를 확인할 수 있습니다.
- `pub.publish(hello_str)`: `chatter` 토픽에 문자열을 퍼블리싱.
- `rate.sleep()`: 지정된 속도를 유지하기 위해 대기.

6. 예외 처리

```
try:
    talker()
except rospy.ROSInterruptException:
    pass
```

`__main__` 체크 외에, `rospy.ROSInterruptException` 예외를 처리합니다. 이 예외는 `Ctrl-C` 입력이나 노드 종료 시 `rospy.sleep()` 또는 `rospy.Rate.sleep()`에서 발생할 수 있습니다. 이를 통해 종료 후 코드가 의도치 않게 실행되는 것을 방지합니다.

- 서브스크라이버 노드 작성

이제 메시지를 수신하는 노드("listener")를 작성해 보겠습니다.

- 코드 작성

`listener.py` 파일을 `scripts` 디렉토리에 다운로드:

```
$ roscd beginner_tutorials/scripts/
$ wget <https://raw.github.com/ros/ros_tutorials/kinetic-devel/rospy_tutorials/001_talker_listener/listener.py>
$ chmod +x listener.py
```

- `listener.py` 코드

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
```

```
def listener():
    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

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

`CMakeLists.txt`의 `catkin_install_python()` 호출을 수정하여 두 스크립트를 모두 포함:

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

- 코드 설명

`listener.py`는 `talker.py`와 유사하지만, 메시지 수신을 위한 콜백 기반 메커니즘을 추가했습니다.

1. 노드 초기화 및 구독 설정

```
rospy.init_node('listener', anonymous=True)
rospy.Subscriber("chatter", String, callback)
rospy.spin()
```

- `rospy.init_node('listener', anonymous=True)`: 노드 이름을 `listener`로 설정하고, `anonymous=True`로 고유 이름을 보장합니다. ROS는 동일한 이름의 노드가 실행되면 이전 노드를 종료하므로, 이 플래그는 여러 `listener` 노드를 동시에 실행할 수 있게 합니다.
- `rospy.Subscriber("chatter", String, callback)`: `chatter` 토픽을 구독하며, `String` 타입 메시지를 수신할 때 `callback` 함수를 호출합니다.

- `rospy.spin()`: 노드가 종료될 때까지 Python이 종료되지 않도록 유지합니다. `roscpp`와 달리, `rospy.spin()`은 구독자 콜백 함수에 영향을 주지 않으며, 콜백은 별도의 스레드에서 처리됩니다.

2. 콜백 함수

```
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
```

새 메시지가 수신되면 `callback` 함수가 호출되며, 메시지는 첫 번째 인수로 전달됩니다. 여기서는 노드의 호출자 ID와 수신된 메시지(`data.data`)를 로그로 출력합니다.

- 노드 빌드

ROS는 CMake를 빌드 시스템으로 사용하며, Python 노드도 CMake를 사용해야 합니다. 이는 메시지와 서비스에 대한 자동 생성 Python 코드를 생성하기 위함입니다.

catkin 작업 공간으로 이동하여 `catkin_make` 실행:

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