Table 07.1: built-in ROS field- and constant-types for messages.

type	serialization	C++	Python 2/3
`bool`	unsigned 8-bit int	**`uint8_t`**	`bool`
`int8`	signed 8-bit int	**`int8_t`**	`int`
`uint8`	unsigned 8-bit int	**`uint8_t`**	`int`
`int16`	signed 16-bit int	**`int16_t`**	`int`
`uint16`	unsigned 16-bit int	**`uint16_t`**	`int`
`int32`	signed 32-bit int	**`int32_t`**	`int`
`uint32`	unsigned 32-bit int	**`uint32_t`**	`int`
`int64`	signed 64-bit int	**`int64_t`**	`long`/`int`
`uint64`	unsigned 64-bit int	**`uint64_t`**	`long`/`int`
`float32`	32-bit IEEE float	**`float`**	`float`
`float64`	64-bit IEEE float	**`double`**	`float`
`string`	ascii string	`std::string`	`str`/`bytes`
`time`	sec/nsec unsigned 32-bit int	`ros::Time`	`rospy.Time`
`duration`	sec/nsec signed 32-bit int	`ros::Duration`	`rospy.Duration`

Lecture 07.03 Custom messages

The messages that come in the `std_msgs` should be exhausted before considering the specification of a new *message description*: a line-separated list of *field* type-name pairs and *constant* type-name-value triples. For example, the following is a message description with two fields and a constant.

message description
field
constant

```
int32 x      # field type: int32, name: x
float32 y    # field type: float32, name: y
int32 Z      # constant type: int32, name: Z
```

The field- and constant-types are usually ROS built-in types, which are shown in Table 07.1. Other field- and constant-types are possible, as described in the documentation:

`wiki.ros.org/msg`.

Of particular interest are arrays of built-in types, like the variable-length array of integers `int32[] foo`, which is interpreted as a Python `tuple`.

To use a custom message description, create a *`.msg` file* in the subdirectory `<package>/msg/` (you may need to create the subdirectory) and enter your message description.

`.msg` file

07.03.1 An example message description

In this section, we develop a custom message description `Complex` in `msg/Complex.msg` for messages with a real and an imaginary floating-point number. We continue to build on the package we've been creating in this chapter, `my_topics`, which shadows the package included with the book, `rico_topics`.

The first thing when creating a custom message description is to create the message description file.

07.03.1.1 Creating a message description

From your package root, create it with the following.

```
mkdir msg
touch msg/Complex.msg
```

Now we can edit the contents of `Complex.msg` to include the following.

```
float32 real
float32 imaginary
```

Both field types are `float32` and have field names `real` and `imaginary`.

We are now ready to update the build-system

07.03.1.2 Updating the build-system configuration

The package we've been working on in this chapter, `my_topics`, was created with a bit of forethought: we included as dependencies in our `catkin_create_pkg` call the packages `message_runtime` and `message_generation`. If we hadn't had such foresight, we would have to make several changes in our package's `package.xml` and `CMakeLists.txt` files before proceeding to create our own message description. As it stands, we still need to make a few changes to them.

How we need to change `package.xml`

Including `message_runtime` and `message_generation` in our `catkin_create_pkg` call yielded the following lines in our `package.xml`, which would otherwise need to be added manually.

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

However, we still need to add `message_runtime` as a **`<build_depend>`**.

```
<build_depend>message_runtime</build_depend>
```

How we need to change `CMakeLists.txt`

Including `message_runtime` and `message_generation` in our `catkin_create_pkg` call yielded the following lines in our `CMakeLists.txt`, which would otherwise need to be added manually. As an additional line in the `find_package(...)` block, we would need the following.

```
message_generation
```

The rest of the changes we do need to make manually. As an additional line in the `catkin_package(...)` block, we need the following.

```
CATKIN_DEPENDS message_runtime
```

The `add_message_files(...)` block needs uncommented and edited to appear as follows.

```
add_message_files(
  FILES
  Complex.msg
)
```

We have already created the `Complex.msg` file.

Finally, the `generate_messages(...)` block needs to be uncommented such that it appears as follows.

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Now our package is set up to use the message type `Complex`—or, it will be once we `catkin_make` our workspace. First, let's write a simple publisher and subscriber to try it out.

```python
#!/usr/bin/env python
import rospy
from rico_topics.msg import Complex # custom message type
from random import random # for random numbers!

rospy.init_node('message_publisher') # initialize node

pub = rospy.Publisher(    # register topic
  'complex',              # topic name
  Complex,                # custom message type
  queue_size=3            # queue size
)

rate = rospy.Rate(2) # set rate

while not rospy.is_shutdown(): # loop
  msg = Complex()            # declare type
  msg.real = random()        # assign value
  msg.imaginary = random()   # assign value

  pub.publish(msg)  # publish!
  rate.sleep()      # sleep to keep rate
```

Figure 07.3: `rico_topics/src/message_publisher.py` listing.

07.03.1.3 Writing a publisher and subscriber

We can now write a publisher and subscriber that publish and subscribe to messages with type `Complex`. Create (`touch`) a Python node file `my_topics/src/message_publisher.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_topics/src/message_publisher.py` file shown in Figure 07.3.

Repeat a similar process to create a `my_topics/src/message_subscriber.py` with the same contents as the `rico_topics/src/message_subscriber.py` file shown in Figure 07.4.

07.03.1.4 Running and verifying these nodes

Let's try it out. Navigate to your workspace root and build your workspace.

```
catkin_make
```

```
#!/usr/bin/env python
import rospy
from rico_topics.msg import Complex

def callback(msg):
    print 'Real:', msg.real           # print real part
    print 'Imaginary:', msg.imaginary # print imag part
    print                             # blank line

rospy.init_node('message_subscriber') # initialize node

sub = rospy.Subscriber( # register subscription
  'complex',            # topic
  Complex,              # custom type
  callback              # callback function
)

rospy.spin() # keep node running until shut down
```

Figure 07.4: `rico_topics/src/message_subscriber.py` listing.

Fire up a `roscore`. In a new Terminal, in your workspace root, `source devel/setup.bash` then run the publisher node.

```
rosrun my_topics message_publisher.py
```

In another new Terminal, in your workspace root, again `source devel/setup.bash` then run the subscriber node.

```
rosrun my_topics message_subscriber.py
```

```
Real: 0.308157861233
Imaginary: 0.229206711054

Real: 0.121079094708
Imaginary: 0.568501293659

Real: 0.807860195637
Imaginary: 0.486804276705
```

It works! Random complex numbers are being printed by the `message_subscriber.py` node.