Part 3: Bot.py Module

Writing our `bot.py` module.

Bot Setup

First, within your `network_project/talkback` directory, create the following file:

```
(NetworkProj) $ touch bot.py
```

`bot.py` should be in the same directory as `quote_picker.py` from the previous section. Go ahead and open up `bot.py` within your text editor.

With `bot.py`, we only need to leverage modules from the Twisted library. There's no expectation that you would know which modules from Twisted to import; this is just an introduction to the package's vast capabilities in networking. In this package, we are taking advantage of Twisted's `log` module for logging rather than using Python's `logging` module, `protocol` module to create our bot factory (to be explained), as well as leverage Twisted's `irc` module so we don't reinvent the wheel.

Note that the order of import statements are alphabetical per PEP-8 (http://www.python.org/dev/peps/pep-0008/), Python's style guide.

```
from twisted.internet import protocol
from twisted.python import log
from twisted.words.protocols import irc
```

Scaffolding for bot.py module

We will write two classes: `TalkBackBot` and `TalkBackBotFactory`. The factory class actually instantiates the bot, while the bot class defines the bot's behavior.

Let's first start off with the bot factory scaffolding with comments and docstrings:

```python
# <--snip-->
class TalkBackBotFactory(protocol.ClientFactory):
    # instantiate the TalkBackBot IRC protocol

    def __init__(self, settings):
        """Initialize the bot factory with our settings."""
```

The factory is in charge of creating/instantiating a protocol (here, the `TalkBackBot`). With the bot factory, we inherit from Twisted's `protocol.ClientFactory` . This is so we can make use of creating a connection between our client and the protocol (our IRC connection), and handle any connection errors.

Now our `TalkBackBot` scaffolding:

```python
# <--snip-->

class TalkBackBot(irc.IRCClient):

    def connectionMade(self):
        """Called when a connection is made."""

    def connectionLost(self, reason):
        """Called when a connection is lost."""

    # callbacks for events

    def signedOn(self):
        """Called when bot has successfully signed on to server."""


    def joined(self, channel):
        """Called when the bot joins the channel."""


    def privmsg(self, user, channel, msg):
        """Called when the bot receives a message."""


# <--snip-->
```

The `TalkBackBot` class inherits from `irc.IRCClient` from the Twisted library. This is so we can make use of functions like `connectionMade` , `signedOn` , etc, and define desired behavior.

First, we'll code out the bot factory, then return to the bot itself.

TalkBackBotFactory class

We first define the protocol that the Factory will make the bot with:

```
# <--snip-->

protocol = TalkBackBot

# <--snip-->
```

This calls an internal method within the `twisted.internet.protocol` library, `buildProtocol()`. This instantiates a `ClientFactory` to be able to handle input of an incoming server connection.

Notice that in our import statements, we didn't import our `settings.ini` file. When we run our program, the plugin that we write (detailed in Part 4 (/networks/part-4)) will pick up the file. With that, our `TalkBackBotFactory` initializes with the settings:

```
# <--snip-->

def __init__(self, channel, nickname, realname, quotes, triggers):
    """Initialize the bot factory with our settings."""
    self.channel = channel
    self.nickname = nickname
    self.realname = realname
    self.quotes = quotes
    self.triggers = triggers

# <--snip-->
```

The initialization of our factory is pretty self explanatory – the factory is created with settings that are defined in `settings.ini`. When we write our plugin in part 4 (/networks/part-4), we will code out the passing of those configuration settings into our factory.

TalkBackBot class

Now for the `TalkBackBot` class. Revisiting the scaffolding we did earlier, we define 5 functions for our class, which will setup the behavior for our bot:

```
# <--snip-->

class TalkBackBot(irc.IRCClient):

    def connectionMade(self):
        """Called when a connection is made."""

    def connectionLost(self, reason):
        """Called when a connection is lost."""

    # callbacks for events

    def signedOn(self):
        """Called when bot has successfully signed on to server."""


    def joined(self, channel):
        """Called when the bot joins the channel."""


    def privmsg(self, user, channel, msg):
        """Called when the bot receives a message."""


# <--snip-->
```

First, the `connectionMade` function: this is considered the initialization of the protocol because it is called when the connection from our client to the IRC server is completed.

```
# <--snip-->

def connectionMade(self):
    """Called when a connection is made."""
    self.nickname = self.factory.nickname
    self.realname = self.factory.realname
    irc.IRCClient.connectionMade(self)
    log.msg("connectionMade")

# <--snip-->
```

When we connect to the IRC service (which we will code out in Part 4), want to assign the `nickname` and `realname` of the bot. If we wanted a greeting message upon connecting to IRC, we would also define it here.

We then call `irc.IRCClient.connectionMade`, which takes in the whole TalkBackBot object (`self`), which contains the `nickname` and `realname` variables.

Lastly, we wish to log this action. We are taking advantage of Twisted's `log` module, which will take care of the time stamps for when each `log.msg()` function is called. We pass in the string `"connectionMade"` so when we consult our logs, we can see this function was called and a connection was made. This is very helpful for debugging purposes. If we were having issues with connecting to the IRC server, we would not hit this log message, and therefore narrow down where the issue is.

Our next function, `connectionLost`, is very similiar:

```
# <--snip-->

def connectionLost(self, reason):
    """Called when a connection is lost."""
    irc.IRCClient.connectionLost(self, reason)
    log.msg("connectionLost {!r}".format(reason))

# <--snip-->
```

`connectionLost` is called when the connection to the IRC Server is closed and "tears down" our protocol. Here, we log the action and the reason.

In our `log.msg()` line, we pass in a string, `"connection lost, reconnecting {!r}"` followed by the string method, `format`. The curly braces, `{}`, indicate a replacement field. This field will be populated by `reason`, an argument passed into our `connectionLost` function.

The `!r` tells `format` to call the function `repr()` (rather than the `str()` function) on `reason`. If we were to do `!s` instead, `format` would *not* include quotes around `reason`, like so (http://docs.python.org/2/library/string.html#formatexamples):

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

To understand `repr()` versus `str()` better, StackOverflow (http://stackoverflow.com/questions/1436703/difference-between-str-and-repr-in-python) has a great explanation.

Next, we define our `signedOn` function:

```python
# <--snip-->

def signedOn(self):
    """Called when bot has successfully signed on to server."""
    log.msg("Signed on")
    if self.nickname != self.factory.nickname:
        log.msg('Your nickname was already occupied, actual nickname is '
                '"{}".'.format(self.nickname))
    self.join(self.factory.channel)

# <--snip-->
```

This function will be called when our bot has successfully signed on to our IRC server. This is different than simply connecting to the IRC server; `connectionMade` is to be treated as the initialization/setup of our protocol (IRC), and `signedOn` is called when we have successfully signed on with our nickname to the server.

The log message is pretty self-explanatory; we are simply logging the action of signing on.

We also put logic in case there is another user or bot signed on with the same nickname. When connecting to an IRC server and your nickname is already in use, the server will often give you a modified nickname, like `thatswhatshesaid_` with the trailing `_`.

Last, we call the `self.join()` function, defined in `irc.IRCClient`, to actually join our desired channel.

Our next function, `joined`, is called after the event of when the bot joins our desired channel. We are simply logging our actions here:

```python
# <--snip-->

def joined(self, channel):
    """Called when the bot joins the channel."""
    log.msg("[{nick} has joined {channel}]"
            .format(nick=self.nickname, channel=self.factory.channel,))

# <--snip-->
```

Our last function for our `TalkBackBot` class is the fun part: defining what happens when someone says “that’s what she said” in our channel:

```python
# <--snip-->

def privmsg(self, user, channel, msg):
    """Called when the bot receives a message."""
    sendTo = None
    prefix = ''
    senderNick = user.split('!', 1)[0]
    if channel == self.nickname:
        # /MSG back
        sendTo = senderNick
    elif msg.startswith(self.nickname):
        # Reply back on the channel
        sendTo = channel
        prefix = senderNick + ': '
    else:
        msg = msg.lower()
        for trigger in self.factory.triggers:
            if msg in trigger:
                sendTo = channel
                prefix = senderNick + ': '
                break

    if sendTo:
        quote = self.factory.quotes.pick()
        self.msg(sendTo, prefix + quote)
        log.msg(
            "sent message to {receiver}, triggered by {sender}:\n\t{quote}"
            .format(receiver=sendTo, sender=senderNick, quote=quote)
        )

# <--snip-->
```

The `privmsg` is called whenever the bot receives a message. We first initialize who we are replying to, `sendTo = None`, and the prefix for our eventual message, `prefix = ''`, as well as `senderNick` who is the user who prompts the bot with the trigger.

The `if channel == self.nickname` is for the condition when the bot receives a message directly, like with `/msg`. The second condition, `elif msg.startswith(self.nickname)` is for when a user starts a message with the bot's nickname within the channel. The last is if someone in the channel says the trigger, "That's what she said."

Basically, if the bot receives a private message, gets mentioned in the beginning of a message, or if someone says a trigger, we set `sendTo` from `None` to the appropriate reply.

Then, if `sendTo` isn't anything but `None`, we construct a quote by picking our quote at random, then using `self.msg` (which we can use because the `msg` method is defined in `irc.IRCClient`) to execute the sending of the message. Lastly, we log our action.

init.py

Within `network/talkback/` directory, you'll notice that there is an empty `__init__.py` file. It is used to mark directories that are a part of our Python package/application. According to Python's documentation for packages (http://docs.python.org/2/tutorial/modules.html#packages):

> The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

← Part 2: Quote Picker (/networks/part-2/)

Part 4: Twisted Plugin → (/networks/part-4/)