Python

IRC on a Higher Level
Contributed by Peyton McCullough
2005−10−05

Python is very suitable for working with the Internet Relay Chat (IRC) protocol. But working directly with a protocol can be a bit messy. Fortunately, there is a library that can simplify this work: Python−IRCLib. This article, the first of three parts, takes a look at this library and what you can do with it, with a focus on event handling.

Introduction

I have written several articles for DevShed relating to Internet Relay Chat – IRC. It's an area of personal interest, and I think Python is very suitable for working with the protocol. However, the articles have focused on working directly with the protocol, rather than employing a library to handle the protocol work. If you have read the articles, you probably know that protocol work can be a bit messy.

Thankfully, a library exists that can work in the middle of your program and the IRC protocol, Python−IRCLib. The library greatly simplifies the creation of applications that interact through IRC by abstracting the protocol. It even contains code that deals specifically with the creation of bots that can communicate with and take orders from the members of the specified IRC server. What's more is that the bots are incredibly simple to create.

The library currently has no documentation. It only contains brief examples of applications that use it. That's a shame, though, since it is a great library. However, in this article, we will examine the library through both explanation and example.

The library may be obtained here:

http://sourceforge.net/projects/python−irclib

Before you start coding, join the network “irc.freenode.net” and enter the channel “#irclib” on your IRC client of preference. We'll do all of our testing there. If you wish, though, you are free to use any channel on any network. However, it is important that you do not test the code in this article out in a populated channel. Users will quickly get annoyed.

Now it's time to jump into some code. First, we'll try joining an IRC server, sending it a message and leaving. Python−IRCLib makes this quite easy:

```
import irclib

# Connection information
network = 'irc.freenode.net'
port = 6667
channel = '#irclib'
```

```
nick = 'PyTest'
name = 'Python Test'

# Create an IRC object
irc = irclib.IRC()

# Create a server object, connect and join the channel
server = irc.server()
server.connect ( network, port, nick, ircname = name )
server.join ( channel )

# Jump into an infinite loop
irc.process_forever()
```

It may take a moment for your application to join the channel, so don't panic if you have to wait a moment. Notice how much easier it is to use a library rather than work directly with the IRC protocol.

With a single IRC object, it is possible to create multiple server objects, allowing for multiple connections:

```
import irclib

# Connection information for the first connection
network1 = 'irc.freenode.net'
port1 = 6667
channel1 = '#irclib'
nick1 = 'PyTest1'
name1 = 'Test One'

# Information for the second connection
network2 = 'irc.freenode.net'
port2 = 6667
channel2 = '#irclib'
nick2 = 'PyTest2'
name2 = 'Test Two'

# Create an IRC object
irc = irclib.IRC()

# Make the first connection
server1 = irc.server()
server1.connect ( network1, port1, nick1, ircname = name1 )
server1.join ( channel1 )

# Make the second connection
server2 = irc.server()
server2.connect ( network2, port2, nick2, ircname = name2 )
server2.join ( channel2 )
```

```
# Infinite loop
irc.process_forever()
```

The *privmsg* method is used to send messages. Recall that “PRIVMSG” is used to send messages to both users and channels:

```
import irclib

# Set this variable to your nickname
me = 'Peyton'

# Connection information
network = 'irc.freenode.net'
port = 6667
channel = '#irclib'
nick = 'PyTest'
name = 'Test One'

# Connect
irc = irclib.IRC()
server = irc.server()
server.connect ( network, port, nick, ircname = name )
server.join ( channel )

# Message both the channel and you
server.privmsg ( channel, 'PRIVMSG to a channel.' )
server.privmsg ( me, 'PRIVMSG to a user.' )

# Loop
irc.process_forever()
```

Now that you know how to connect your application to an IRC network, you are probably wondering how to do something useful, namely how to process incoming information and respond appropriately to it. Before we get to that, though, it is important that you understand exactly what information is coming to you. Python−IRCLib will display information to and from the server if you simply add this line after you import *irclib*:

```
irclib.DEBUG = True
```

Try creating a program with the line and examining the incoming and outgoing data.

Now we need to handle the data. Let's work on handling the data initially received from the server first. To do this, we register an event handler with the add_global_handler, passing two or three arguments. The first argument is the code that corresponds to the event we want to handle. We'll get to this in a second. The second argument points to the function that is to handle the event. The third argument is optional, and it is named *priority*. The lower the number is, the higher the priority of the event:

```
import irclib
```

```
# Connection information
network = 'irc.freenode.net'
port = 6667
channel = '#irclib'
nick = 'PyTest'
name = 'Test One'

# Generic echo handler ( space added )
# This is used to output some initial information from the
server
def handleEcho ( connection, event ):

   print
   print ' '.join ( event.arguments() )

# Generic echo handler ( no space added )
def handleNoSpace ( connection, event ):

   print ' '.join ( event.arguments() )

# Handle private notices
def handlePrivNotice ( connection, event ):

   if event.source():
      print ':: ' + event.source() + ' ->' + event.arguments()
[ 0 ]
   else:
      print event.arguments() [ 0 ]

# Handle channel joins
def handleJoin ( connection, event ):

   # The source needs to be split into just the name
   # It comes in the format nickname!user@host
   print event.source().split ( '!' ) [ 0 ] + ' has joined ' +
event.target()

# Create the IRC object
irc = irclib.IRC()

# Register handlers
irc.add_global_handler ( 'privnotice', handlePrivNotice ) #
Private notice
irc.add_global_handler ( 'welcome', handleEcho ) # Welcome
message
irc.add_global_handler ( 'yourhost', handleEcho ) # Host
message
irc.add_global_handler ( 'created', handleEcho ) # Server
creation message
irc.add_global_handler ( 'myinfo', handleEcho ) # "My info"
message
```

```
irc.add_global_handler ( 'featurelist', handleEcho ) # Server
feature list
irc.add_global_handler ( 'luserclient', handleEcho ) # User
count
irc.add_global_handler ( 'luserop', handleEcho ) # Operator
count
irc.add_global_handler ( 'luserchannels', handleEcho ) #
Channel
count
irc.add_global_handler ( 'luserme', handleEcho ) # Server
client
count
irc.add_global_handler ( 'n_local', handleEcho ) # Server
client
count/maximum
irc.add_global_handler ( 'n_global', handleEcho ) # Network
client count/maximum
irc.add_global_handler ( 'luserconns', handleEcho ) # Record
client count
irc.add_global_handler ( 'luserunknown', handleEcho ) #
Unknown
connections
irc.add_global_handler ( 'motdstart', handleEcho ) # Message
of
the day ( start )
irc.add_global_handler ( 'motd', handleNoSpace ) # Message of
the
day
irc.add_global_handler ( 'edofmotd', handleEcho ) # Message of
the day ( end )
irc.add_global_handler ( 'join', handleJoin ) # Channel join
irc.add_global_handler ( 'namreply', handleNoSpace ) # Channel
name list
irc.add_global_handler ( 'endofnames', handleNoSpace ) #
Channel
name list ( end )

# Connect to the network
server = irc.server()
server.connect ( network, port, nick, ircname = name )
server.join ( channel )

# Run an infinite loop
irc.process_forever()
```

The above code may seem like a lot, but in reality, it's not. Before we create the IRC object, we define a few functions to handle certain events. Two are generic functions that merely echo the arguments of the received commands, and the other two are aimed at specific events. We then create the IRC object and register handlers for a number of events.

The first events we handle are private notices. The server will issue several of these upon connection, so we need to find a way to display them. We then handle a number of messages that the server gives us upon connection. The messages mainly deal with statistics and features of the server, so I won't get into too much detail. They are pretty easy to analyze.

When the server is about to send us the message of the day, it sends us the command “375”. Python−IRCLib translates this to “motdstart”. We'll be handling many numeric commands from the server, so I'll provide a list shortly. If the *DEBUG* variable is set to *True*, the library will display the translated command to us, too. The server then sends us the message of the day, followed by a command to end the message of the day.

Our client is set to join the channel “#irclib”. When this happens, a message will be sent to the channel notifying its users of our presence. The function *handleJoin* handles this event. We simply take the source of the command and join it to the target of the command. We also break apart the source to get a more readable name. When we join the channel, the server sends a list of names to us, which we display with *handleNoSpace*.

The handler functions are passed two arguments, an event object and a server object. The former contains the source of the command (*event.source*), the target of the command (*event.target*) and the arguments of the command (*event.arguments*). The latter is used in case we need to respond to the message sent. For example, let's consider an “INVITE” message from another user. It is simply a message from the user inviting us to join a certain channel. We could set up a handler that would automatically join the channel like this:

```
def handleInvite ( connection, event ):

   connection.join ( event.arguments() [ 0 ] )

...

irc.add_global_handler ( 'invite', handleInvite ) # Invite
```

Our application is, of course, far from complete if we're looking to handle even the more common events. We still have more events to consider. For example, if someone sends a message to use or the channel, it would be wise to display or consider a reply:

```
# Private messages
def handlePrivMessage ( connection, event ):

   print event.source().split ( '!' ) [ 0 ] + ': ' +
event.arguments() [ 0 ]

   # Respond to a “hello” message
   if event.arguments() [ 0 ].lower().find ( 'hello' ) == 0:
      connection.privmsg ( event.source().split ( '!' ) [ 0 ],
'Hello.' )

# Public messages
def handlePubMessage ( connection, event ):
   print event.target() + '> ' + event.source().split ( '!' )
[ 0 ] + ': ' +
event.arguments() [ 0 ]

...

irc.add_global_handler ( 'privmsg', handlePrivMessage )
irc.add_global_handler ( 'pubmsg', handlePubMessage )
```

Another common event would be someone changing the topic of a channel. This event could easily be caught:

```
def handleTopic ( connection, event ):

   print event.source().split ( '!' ) [ 0 ] + ' has set the
topic
to "' + event.arguments() [ 0 ] + '"'

...

irc.add_global_handler ( 'topic', handleTopic )
```

Another common event would be a mode change to either a channel or a user. This bit of code will catch the event and display the appropriate message:

```
def handleMode ( connection, event ):

   # Channel mode
   if len ( event.arguments() ) < 2:
      print event.source().split ( '!' ) [ 0 ] + ' has altered
the channel\'s mode: ' + event.arguments() [ 0 ]

   # User mode
   else:
      print event.source().split ( '!' ) [ 0 ] + ' has altered
'
+ event.arguments() [ 1 ] + '\'s mode: ' + event.arguments()
[ 0 ]

...

irc.add_global_handler ( 'mode', handleMode )
```

Of course, the users of a network will leave channels, and we should learn how to catch such an event. A user may part a channel, disconnect from the network or be removed from the channel by an operator, so we should consider all three events:

```
def handlePart ( connection, event ):

   print event.source().split ( '!' ) [ 0 ] + ' has quit ' +
event.target()
```

```
def handleQuit ( connection, event ):

   print event.source().split ( '!' ) [ 0 ] + ' has
disconnected:
' + event.arguments() [ 0 ]

def handleKick ( connection, event ):

   print event.arguments() [ 0 ] + ' has been kicked by ' +
event.source().split ( '!' ) [ 0 ] + ': ' + event.arguments()
[ 1 ]

...

irc.add_global_handler ( 'part', handlePart )
irc.add_global_handler ( 'quit', handleQuit )
irc.add_global_handler ( 'kick', handleKick )
```

Now our application should be sensitive to a variety of common events. Each handler function can be modified to perform an appropriate action. For examle, if you wanted to welcome users to the channel when they join, you would simply modify whatever function is attached to the event “join”.

Sometimes, you'll want to unbind a function. For example, let's say you wanted to remove *handleKick* from the “kick” event. You would call the *remove_global_handler* method, which takes the same arguments as *add_global_handler*:

```
irc.remove_global_handler ( 'kick', handleKick )
```

Earlier, I promised a list of events that you can respond to. Here is a list of numeric codes and their appropriate codes:

```
216 -> statskline
217 -> statsqline
214 -> statsnline
215 -> statsiline
212 -> statscommands
213 -> statscline
210 -> tracereconnect
211 -> statslinkinfo
218 -> statsyline
219 -> endofstats
491 -> nooperhost
492 -> noservicehost
407 -> toomanytargets
406 -> wasnosuchnick
346 -> invitelist
347 -> endofinvitelist
403 -> nosuchchannel
341 -> inviting
342 -> summoning
```

```
348 -> exceptlist
349 -> endofexceptlist
409 -> noorigin
263 -> tryagain
262 -> endoftrace
261 -> tracelog
266 -> n_global
265 -> n_local
442 -> notonchannel
423 -> noadmininfo
422 -> nomotd
424 -> fileerror
414 -> wildtoplevel
437 -> unavailresource
411 -> norecipient
412 -> notexttosend
413 -> notoplevel
371 -> info
373 -> infostart
372 -> motd
375 -> motdstart
374 -> endofinfo
377 -> motd2
376 -> endofmotd
319 -> whoischannels
318 -> endofwhois
313 -> whoisoperator
312 -> whoisserver
311 -> whoisuser
317 -> whoisidle
316 -> whoischanop
315 -> endofwho
314 -> whowasuser
393 -> users
392 -> usersstart
391 -> time
395 -> nousers
394 -> endofusers
443 -> useronchannel
368 -> endofbanlist
369 -> endofwhowas
366 -> endofnames
367 -> banlist
364 -> links
365 -> endoflinks
362 -> closing
363 -> closeend
361 -> killdone
300 -> none
301 -> away
302 -> userhost
```

```
303 -> ison
305 -> unaway
306 -> nowaway
444 -> nologin
244 -> statshline
382 -> rehashing
241 -> statslline
445 -> summondisabled
243 -> statsoline
242 -> statsuptime
381 -> youreoper
436 -> nickcollision
384 -> myportis
432 -> erroneusnickname
433 -> nicknameinuse
431 -> nonicknamegiven
451 -> notregistered
331 -> notopic
333 -> topicinfo
332 -> topic
258 -> adminloc2
259 -> adminemail
252 -> luserop
253 -> luserunknown
250 -> luserconns
251 -> luserclient
256 -> adminme
257 -> adminloc1
254 -> luserchannels
255 -> luserme
405 -> toomanychannels
404 -> cannotsendtochan
502 -> usersdontmatch
402 -> nosuchserver
401 -> nosuchnick
465 -> yourebannedcreep
464 -> passwdmismatch
467 -> keyset
466 -> youwillbebanned
461 -> needmoreparams
463 -> nopermforhost
462 -> alreadyregistered
221 -> umodeis
446 -> usersdisabled
501 -> umodeunknownflag
234 -> servlist
235 -> servlistend
231 -> serviceinfo
232 -> endofservices
233 -> service
441 -> usernotinchannel
```

```
322 −> list
323 −> listend
321 −> liststart
324 −> channelmodeis
476 −> badchanmask
329 −> channelcreate
477 −> nochanmodes
201 −> traceconnecting
200 −> tracelink
203 −> traceunknown
202 −> tracehandshake
205 −> traceuser
204 −> traceoperator
207 −> traceservice
206 −> traceserver
209 −> traceclass
208 −> tracenewtype
475 −> badchannelkey
003 −> created
002 −> yourhost
001 −> welcome
005 −> featurelist
004 −> myinfo
474 −> bannedfromchan
485 −> uniqopprivsneeded
484 −> restricted
483 −> cantkillserver
482 −> chanoprivsneeded
481 −> noprivileges
472 −> unknownmode
473 −> inviteonlychan
471 −> channelisfull
353 −> namreply
352 −> whoreply
351 −> version
421 −> unknowncommand
478 −> banlistfull
```

As you can see, it's quite a list. Specific information can be found in the IRC protocol's documentation, but you most likely won't want to respond to most of the events supported by Python−IRCLib. There is simply no need, unless your aim is to create a full IRC client.

A few more events are also defined:

```
dcc_connect
dcc_disconnect
dccmsg
disconnect
ctcp
ctcpreply
error
```

```
join
kick
mode
part
ping
privmsg
privnotice
pubmsg
pubnotice
quit
```

The most recent events contain mostly events to which you will need to give some thought when building your application, such as private messages and notices.

Conclusion

In this article, we've taken a look at a very important concept in Python−IRCLib: catching and reacting to events. Any application of purpose that interacts through IRC will have to examine events, whether it simply saves statistics and logs of activity or does something a bit more complex, such as take commands from select users and behave appropriately.

It is not possible to cover all of the library's features in just one article, so don't worry if you feel left in the dark right now. An understanding of events is important before we continue to more of the library's features, such as the building of bots.

Experiment with event handling in Python−IRCLib. One helpful technique is to turn the DEBUG variable on and examine the incoming data. If anything pops up that you don't recognize, it might be worth taking a look at.