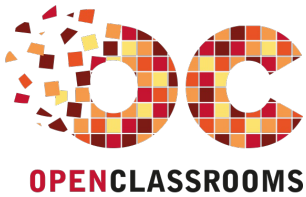
Programmer un bot IRC simplement avec ircbot

Par Dentuk
et Epoch



www.openclassrooms.com

Dernière mise à jour le 1/09/2010

Sommaire

Programmer un bot IRC simplement avec ircbot

L'auteur de ce cours cherche un repreneur pour continuer son travail
Parfois, certains auteurs de cours n'ont plus le temps de rédiger et de mettre à jour leur travail. Ils recherchent alors un nouvel auteur pour "reprendre" leur tutoriel **(voir la liste des tutoriels en attente de repreneur)**.
Contactez l'auteur si vous aimeriez continuer la rédaction !

Par  Dentuk et Epoch
Mise à jour : 01/09/2010
Difficulté : Intermédiaire

Ce tutoriel présente l'utilisation du module ircbot de Joel Rosdahl pour Python. Comme son nom l'indique, celui-ci permet de programmer son propre *bot* IRC. Il utilise le module irclib du même auteur, qui est plus général et qui ne sera que partiellement présenté ici.

Ces deux modules ne sont pour le moment pas compatibles avec Python 3 ! Ce tutoriel requiert donc Python 2 pour être suivi.

Sont prérequises pour suivre ce tutoriel :

- des bases en Python, notamment au niveau de la POO (déclaration d'une classe, héritage...) ;
- et des connaissances sur ce qu'est IRC et les possibilités qu'il offre.

C'est bon ? Alors, suivez le guide.
Sommaire du tutoriel :

Se procurer les fichiers

Les deux modules que nous allons étudier, irclib et ircbot, ont été développés par un programmeur tiers et ne sont donc pas livrés directement avec Python ; vous allez donc devoir vous les procurer avant de commencer !

Le module ircbot utilise irclib. Vous ne pouvez donc pas utiliser ircbot sans avoir installé au préalable irclib.

Si vous êtes sous une distribution Linux gérant les installations par paquets, alors il existe peut-être un paquet python-irclib qui regroupe irclib **et** ircbot. Par exemple, sous Ubuntu, taper `sudo apt-get install python-irclib` installera les modules. Sinon, vous pouvez télécharger la dernière version depuis la page du projet sur SourceForge.net.
Cependant, la version publique des modules gère les *half operators* de la même manière que les utilisateurs normaux ce qui peut produire quelques bugs (utilisateurs qui sont listés avec un % devant leur pseudonyme). Pour cette raison, je vous invite à utiliser une version modifiée par mes soins qui fait bien la distinction ici. Après cela, décompressez l'archive à l'aide de votre logiciel favori. La plupart des fichiers sont des programmes d'exemple, que je vous conseille de regarder plutôt après qu'avant d'avoir lu ce tutoriel.
Maintenant, deux choix s'offrent à vous :

- lancer le script d'installation via setup.py comme indiqué dans le fichier README, de sorte que tous vos codes aient accès à ces deux modules ;
- placer les deux fichiers irclib.py et ircbot.py dans le répertoire du programme pour chaque script utilisant ces modules.

La première solution est conseillée si vous utilisez vos *bots* pour vous-mêmes, puisqu'elle vous évitera de copier les fichiers à chaque fois que vous programmerez un nouveau *bot*. En revanche, si vous comptez distribuer vos programmes, il est fortement recommandé de livrer les fichiers des modules avec, puisque l'utilisateur n'aura pas nécessairement installé irclib et ircbot sur son ordinateur, d'autant plus si vous utilisez la version modifiée. Vous pouvez aussi installer les fichiers pour la période de développement et les distribuer une fois le *bot* terminé.

La programmation évènementielle et ircbot

Avant de commencer à coder, il va falloir faire un brin de théorie. Rien de bien compliqué, bien entendu. 

Si vous utilisez irclib ou ircbot, vous allez devoir faire ce que l'on appelle de la programmation évènementielle. Il s'agit d'une logique de programmation raisonnant sur des **évènements** : quand il se passe telle chose, alors on fait cela. Pour un exemple concret : quand quelqu'un rejoint le canal, alors je lui dis bonjour. Il s'avère que c'est particulièrement adapté à IRC, ce qui explique son utilisation ici.

Dans vos programmes, vous allez définir pour chaque évènement que vous souhaitez traiter une méthode qui sera appelée automatiquement par ircbot lorsque l'évènement en question se produira. Pour cela, vos *bots* devront hériter de la classe ircbot.SingleServerIRCBot.
Chaque méthode devra être de la forme suivante :

Code : Python

```
class MonBot(ircbot.SingleServerIRCBot):
    def on_evenement(self, serv, ev):
        # gestion de l'évènement
```

- Dans le nom de la méthode, evenement sera remplacé par le nom de l'évènement à traiter, ce qui donnera par exemple on_join, on_kick, etc.
- La méthode prend deux paramètres (ici nommés serv et ev). Le premier sert à communiquer avec le serveur, et le second nous renseigne sur l'évènement, nous permettant de savoir qui a été *kické*, par qui et sur quel canal par exemple.

Nous allons maintenant étudier plus en détail ces deux arguments.

Communication avec le serveur

irclib nous permet de faire dialoguer notre *bot* avec le serveur comme le ferait un client IRC habituel à travers la classe irclib.ServerConnection, dont l'argument serv dont nous avons parlé plus haut est une instance. Elle dispose de nombreuses méthodes et seules les plus utilisées seront présentées ici.

Commandes générales

irclib ServerConnection	Utilité
`nick(self, pseudonyme)`	Change le pseudonyme du *bot*.
`get_nickname(self)`	Renvoie votre pseudonyme actuel. Celui-ci n'est pas forcément celui que vous souhaitiez, en particulier si vous demandez un pseudonyme réservé.
`privmsg(self, destinataire, message)`	Envoie un message ; le destinataire peut être soit un canal tel que #kikoo, soit un nom d'utilisateur tel que dentuk.
`action(self, destinataire, message)`	Envoie une action ; sous la plupart des clients, cela correspond à la commande /me.
`notice(self, destinataire, message)`	Envoie une notice.
`invite(self, pseudonyme, canal)`	Invite un autre utilisateur à rejoindre le canal précisé.

`join(self, canal, cle='')`	Rejoint le canal précisé, en utilisant la clé donnée si le canal est protégé par mot de passe.
`part(self, canal, message='')`	Quitte le canal précisé, éventuellement avec un message d'au revoir.
`disconnect(self, message='')`	Quitte le serveur, éventuellement avec un message d'au revoir, et déconnecte le *bot*.

Commandes d'opérateur

irclib ServerConnection	Utilité
`kick(self, canal, pseudonyme, raison='')`	Exclut un utilisateur du canal, éventuellement avec une raison.
`mode(self, canal, commande)`	Active ou retire un mode. Nous verrons comment utiliser les modes plus loin dans le tutoriel.

Voilà déjà de quoi bien vous amuser. Pour la liste complète des méthodes, lancez un *shell* Python et procédez ainsi :

Code : Python

```
>>> import irclib
>>> help(irclib.ServerConnection)
```

Toutefois, la documentation n'est pas très précise, le plus souvent on a droit à un « envoie telle commande au serveur », vous aurez donc peut-être besoin de la compléter par une recherche.

La variable d'évènement

Il y a dans irclib trois catégories d'évènements :

- ceux qui découlent du protocole IRC en lui-même, et sont les plus utiles tels que la réception d'un message ou l'exclusion d'un utilisateur ;
- ceux qui sont générés pour les besoins de irclib, concernant principalement le CTCP et le DCC que nous n'étudierons pas ici ;
- et les évènements dits « numériques », parce qu'ils correspondent à un code numérique dans le protocole IRC dont vous, en tant qu'utilisateur de irclib, n'aurez pas à vous soucier ; ils précisent une étape dans la communication (début / fin d'envoi du message du jour...), une erreur (si l'on essaie d'envoyer un message à un pseudonyme inexistant...) ou autre chose ; ils sont très nombreux et nous n'en étudierons qu'un seul dans ce tutoriel, qui est celui de bienvenue.

Un évènement est représenté par une instance de la classe irclib.Event, c'est le cas de l'argument ev de notre méthode modèle. Cette classe possède trois méthodes d'information qui nous seront utiles, et une autre qui, en général, ne sert pas quand on utilise ircbot.

irclib Event	Renseigne sur...
`source(self)`	... l'origine de l'évènement ; c'est-à-dire, selon l'évènement, soit un serveur, soit un utilisateur, soit un canal.
`target(self)`	... la cible de l'évènement ; c'est-à-dire, selon l'évènement, soit un utilisateur, soit un canal.
`arguments(self)`	... divers paramètres relatifs à l'évènement, toujours sous la forme d'une liste.
`eventtype(self)`	... le type de l'évènement ; par exemple, `"welcome"` pour l'évènement de bienvenue.

Nous allons maintenant voir quelques évènements que vous pouvez choisir de traiter. Bien entendu, vous n'avez pas besoin de tout savoir par coeur, vous pouvez toujours garder les tableaux de côté quand vous programmez. Je vous propose deux tableaux : dans le premier, il y a ce que représente la valeur de retour de chaque méthode de irclib.Event pour chaque évènement ; et dans l'autre, des exemples de ces valeurs de retour pour bien comprendre.

Tableau des évènements

Émis quand...	ev.eventtype()	ev.source()	ev.target()	ev.arguments()
... vous êtes connectés au serveur.	welcome	Serveur	Votre pseudonyme	Message d'accueil
... quelqu'un change de pseudonyme.	nick	Utilisateur	Nouveau pseudonyme	
... quelqu'un vous envoie un message privé.	privmsg	Utilisateur	Votre pseudonyme	Message envoyé
... quelqu'un envoie un message sur le canal.	pubmsg	Utilisateur	Canal	Message envoyé
... quelqu'un utilise la commande /me sur le canal.	action	Utilisateur	Canal	Message d'action
... quelqu'un vous envoie une notice.	privnotice	Utilisateur	Votre pseudonyme	Message envoyé
... quelqu'un envoie une notice au canal.	pubnotice	Utilisateur	Canal	Message envoyé
... quelqu'un invite une autre personne à rejoindre le canal.	invite	Utilisateur	Pseudonyme invité	Canal
... quelqu'un rejoint le canal.	join	Utilisateur	Canal	
... quelqu'un quitte le canal.	part	Utilisateur	Canal	Message d'au revoir
... quelqu'un quitte le serveur.	quit	Utilisateur		Message d'au revoir
... un opérateur expulse un utilisateur.	kick	Utilisateur	Canal	• Pseudonyme de la victime • Raison du *kick*
... un opérateur applique un ou des modes.	mode	Utilisateur	Canal	• Mode(s) appliqué(s) • Argument éventuel • Argument éventuel • ...

Exemples de valeurs de retour

ev.eventtype()	ev.source()	ev.target()	ev.arguments()
`'welcome'`	`'marseille.fr.epiknet.org'`	`'bottuk'`	`['Welcome to the EpiKnet IRC Network bottuk!~bottuk@[...].net']`
`'nick'`	`'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'`	`'palatuk'`	`[]`
`'privmsg'`	`'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'`	`'bottuk'`	`['salut bottuk, comment va ?']`
`'pubmsg'`	`'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'`	`'#kikoo'`	`['bonjour tout le monde !']`

'action'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'#kikoo'	['est heureux !']
'privnotice'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'bottuk'	['salut bottuk !']
'pubnotice'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'#kikoo'	['je notice tout le canal !']
'invite'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'TomicBomb'	['#kikoo']
'join'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'#kikoo'	[]
'part'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'#kikoo'	['je reviens']
'quit'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	None	['Quit: Konversation terminated!']
'kick'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'#kikoo'	['bouletman', "pas d'insultes, merci !"]
'mode'	'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'	'#kikoo'	['+hh', 'bouh', 'A-dream']

Vous vous demandez peut-être comment exploiter les valeurs de retour de la méthode irclib.Event.source, que j'appellerai des masques pour la suite du tutoriel. Bien que l'on puisse extraire les différentes parties assez facilement avec Python seul, irclib fournit plusieurs fonctions pour procéder à l'extraction. Le tableau ci-dessous utilise un masque de la forme `'pseudonyme!utilisateur@domaine'` .

Fonctions d'extraction

irclib	Origine du nom	Extrait...
nm_to_n(masque)	nickmask to nick	... le pseudonyme.
nm_to_u(masque)	nickmask to user	... l'utilisateur.
nm_to_h(masque)	nickmask to host	... le domaine.
nm_to_uh(masque)	nickmask to userhost	... l'utilisateur et le domaine.

Pour illustrer, je vous propose une petite session dans un *shell* Python. 

Code : Python

```
>>> import irclib
>>> masque = 'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'
>>> irclib.nm_to_n(masque)
'dentuk'
>>> irclib.nm_to_u(masque)
'~dentuk'
>>> irclib.nm_to_h(masque)
'EpiK-BE9687C2.fbx.proxad.net'
>>> irclib.nm_to_uh(masque)
'~dentuk@EpiK-BE9687C2.fbx.proxad.net'
```

Parlons maintenant un peu de la classe ircbot.SingleServerIRCBot.

Étude de la classe mère

Nous allons maintenant voir quelques méthodes de la classe mère qui vous seront utiles pour la plupart de vos *bots*.

Méthode d'initialisation

En voici le prototype :

Code : Python

```
ircbot.SingleServerIRCBot.__init__(self, liste_serveurs, pseudonyme,
nom_reel et intervalle_reconnexion=60)
```

Étudions les paramètres un à un.

- liste_serveurs est... une liste de serveurs ! En effet, la classe utilisée permet de changer de serveur en cours d'exécution, même si nous n'approfondirons pas cette possibilité dans ce tutoriel. Chaque serveur est indiqué par une liste de la forme `[serveur, port, mot_de_passe]` qui renseigne les informations à utiliser sur ce serveur, les autres paramètres étant communs à tous les serveurs. Vous n'êtes pas obligés de préciser un mot de passe.
- pseudonyme et nom_reel ont, je pense, un nom assez précis pour que vous vous doutiez de ce qu'ils renseignent. Mais savez-vous ce qu'est le nom réel ? Il s'agit d'une information que le serveur envoie aux personnes désirant en savoir plus sur vous par la commande /whois.
- intervalle_reconnexion possède aussi un nom équivoque puisqu'il s'agit du temps en secondes pendant lequel le *bot* doit patienter avant d'essayer de se reconnecter s'il est déconnecté. Ce paramètre est facultatif et vaut 60 par défaut.

Lancer le bot

On ne peut plus simple, le lancement se fait par l'appel de la méthode ircbot.SingleServerIRCBot.start qui ne prend aucun paramètre. Notez que cela interrompt l'exécution du script pour se consacrer uniquement au *bot*, comme une boucle infinie. On peut en sortir en levant une exception, par conséquent l'interruption Ctrl-C arrête le *bot*.

Arrêter le bot

La méthode `ircbot.SingleServerIRCBot.die(self, message='')` quitte le serveur, déconnecte le *bot* et met fin à l'exécution du script. C'est en général cette méthode que l'on utilise pour arrêter le *bot*. Si vous avez besoin de poursuivre l'exécution du script après la déconnexion, utilisez plutôt la méthode `ircbot.SingleServerIRCBot.disconnect(self, message='')` suivie de la levée d'une exception que vous intercepterez dans le code principal. Notez que si vous employez cette dernière seule, le *bot* se reconnectera au bout d'un certain temps précisé à l'initialisation.

Répondre aux commandes VERSION

Par défaut, la réponse envoyée à une commande VERSION est `"ircbot.py by Joel Rosdahl <joel@rosdahl.net>"` . Si vous désirez changer cela, vous pouvez réimplémenter la méthode `ircbot.SingleServerIRCBot.get_version(self)` , et lui faire retourner le message de votre choix.

C'est la méthode `ircbot.SingleServerIRCBot.on_ctcp(self)` qui, par défaut, répond automatiquement aux commandes PING et VERSION de cette façon. Pensez-y si vous souhaitez réimplémenter cette méthode.

Application à un bot simpliste

Pour mieux comprendre tout ce que l'on a vu jusqu'ici, je propose de vous montrer un exemple de *bot* codé avec ircbot. Il s'agit d'un *bot* de modération simplifié au maximum, qui exclut les personnes écrivant des insultes dans leurs messages. Analysez bien le code et revoyez au besoin ce qui a déjà été vu, le but étant que vous soyez capables de faire un *bot* vous-mêmes après avoir vu cet exemple.
J'ai commenté les parties du code qui le nécessitaient, les autres parties sont simples et je ne m'étendrai donc pas dessus.

Code : Python

```
#!/usr/bin/env python2
# -*- coding: utf8 -*-

import irclib
import ircbot

class BotModeration(ircbot.SingleServerIRCBot):
    def __init__(self):
        """
```

```
        Constructeur qui pourrait prendre des paramètres dans un "vrai"
        programme.
        """
                ircbot.SingleServerIRCBot.__init__(self,
        [("irc.epiknet.org", 6667)],
                                                   "moderator", "Bot de
        modération réalisé en Python avec ircbot")
                self.insultes = ["con", "pute"] # Liste à agrandir pour un
        "vrai" programme.

            def on_welcome(self, serv, ev):
                """
        Méthode appelée une fois connecté et identifié.
        Notez qu'on ne peut rejoindre les canaux auparavant.
        """
                serv.join("#test-ircbot")

            def on_pubmsg(self, serv, ev):
                """
        Méthode appelée à la réception d'un message, qui exclut son
        expéditeur s'il
        écrit une insulte.
        """
                # Il n'est pas indispensable de passer par des variables
                # Ici elles permettent de clarifier le tout.
                auteur = irclib.nm_to_n(ev.source())
                canal = ev.target()
                message = ev.arguments()[0].lower() # Les insultes sont
        écrites en minuscules.

                for insulte in self.insultes:
                    if insulte in message:
                        serv.kick(canal, auteur, "Les insultes ne sont pas
        autorisées ici !")
                        break

        if __name__ == "__main__":
            BotModeration().start()
```

La gestion des insultes pourrait bien sûr être grandement améliorée, mais ce n'est pas le but de cet exemple. Si vous avez tout compris jusqu'ici, vous devriez déjà être capables de programmer des *bots* intéressants. Si vous estimez ces connaissances suffisantes, vous pouvez arrêter ici le tutoriel. En revanche, si vous souhaitez aller un peu plus loin, passez à la partie suivante.


Allons un peu plus loin

Dans cette partie, nous allons voir diverses choses qui pourraient vous être utiles un jour, sans ordre précis. Si une sous-partie ne vous intéresse pas, passez à la suivante.


Masques de bannissement

Si vous êtes opérateur d'un canal, vous avez probablement déjà vu des masques de bannissement. Par exemple, le masque *tuk!*@* englobe les personnes ayant un pseudo finissant par « tuk », quels que soient leur nom d'utilisateur et leur domaine. irclib fournit une fonction pour vérifier si un utilisateur vérifie un masque ainsi formé, dont voici le prototype :

Code : Python

```
irclib.mask_matches(masque_utilisateur, masque_a_verifier)
```

Cette fonction, à l'instar de la fonction re.match, retourne un objet de *matches* en cas de succès et `None` en cas d'échec. À noter que l'objet de *matches* est inexploitable. En général, on utilise la valeur de retour comme s'il s'agissait d'un booléen, parce que `bool(None)` vaut `False` et `bool(un_objet_de_matches)` vaut `True` .
Toujours pour illustrer, une autre session dans le *shell* Python :

Code : Python

```
>>> import irclib
>>> masque = 'dentuk!~dentuk@EpiK-BE9687C2.fbx.proxad.net'
>>> irclib.mask_matches(masque, '*tuk!*@*')
<_sre.SRE_Match object at 0xb7a6d7c8>
>>> irclib.mask_matches(masque, 'Tomic*!*@*')
>>> bool(irclib.mask_matches(masque, '*tuk!*@*'))
True
>>> bool(irclib.mask_matches(masque, 'Tomic*!*@*'))
False
```

Cela peut, par exemple, servir à faire une commande !quit qui dit au *bot* de partir, mais réservée à son possesseur en identifiant ce dernier par un masque :

Code : Python

```
class Bot(ircbot.SingleServerIRCBot):
    def on_pubmsg(self, serv, ev):
        masque_auteur = ev.source()
        message = ev.arguments()[0].lower()
if message == "!quit" and irclib.mask_matches(masque_auteur,
"*!*@EpiK-BE9687C2.fbx.proxad.net"):
self.die()
```

Exécution délimitée dans le temps

Vous pouvez avoir besoin d'attendre un certain temps avant d'exécuter une action. Par exemple, si vous faites un *bot* de quiz et que vous devez attendre une minute, avant de changer de question si personne n'a trouvé la question, pour éviter que le jeu ne soit bloqué. Mais si vous utilisiez time.sleep, vous bloqueriez l'exécution et irclib ne pourrait pas traiter les nouveaux évènements pendant ce temps. Pour régler ce problème, irclib propose deux méthodes.

Exécution au bout d'un certain temps

Voici la méthode à utiliser :

Code : Python

```
irclib.ServerConnection.execute_delayed(self, temps, fonction,
arguments=())
```

Lorsque vous appelez cette méthode, irclib prend en compte qu'il faudra appeler la fonction fonction dans temps secondes avec les arguments arguments, et fera l'appel le moment voulu. temps peut être un nombre entier ou flottant, et arguments est un tuple contenant les arguments à passer à la fonction, qui est vide par défaut.

Par exemple, si vous étiez sadiques et que vous souhaitiez adapter le code du *bot* de modération, vu plus haut, en mettant un compte à rebours avant l'exclusion, vous pourriez implémenter la méthode on_pubmsg ainsi : 

Code : Python

```
class BotModeration(ircbot.SingleServerIRCBot):
    def on_pubmsg(self, serv, ev):
        auteur = irclib.nm_to_n(ev.source())
        canal = ev.target()
        message = ev.arguments()[0].lower()

        for insulte in self.insultes:
            if insulte in message:
serv.privmsg(canal, "3...")
serv.execute_delayed(1, serv.privmsg, (canal, "2..."))
serv.execute_delayed(2, serv.privmsg, (canal, "1..."))
serv.execute_delayed(3, serv.kick, (canal, auteur, "0 !"))
                break
```

Exécution à un moment précis

Code : Python

```
irclib.ServerConnection.execute_at(self, timestamp, fonction,
arguments=())
```

Très similaire, cette méthode fonctionne exactement de la même manière. Simplement, le premier paramètre est maintenant un *timestamp* tel que ceux retournés par `time.time` et `time.mktime` .

Gestion des canaux

Vous vous rappelez quand je vous ai parlé de la méthode `irclib.ServerConnection.mode(self, canal, mode)` ? Il est temps de voir quelques modes qui peuvent être activés et leur effet.

Les principaux modes de canaux sans paramètre

Mode	Signification
m	Le canal est modéré, seuls les privilégiés, les *half operators* et les opérateurs peuvent y parler.
s	Le canal est secret, il est caché lors d'une commande /whois et il n'est pas présent dans la liste des canaux du serveur.
p	Le canal est protégé, ce qui a le même effet qu'être secret.
i	Le canal est accessible uniquement sur invitation.

Si par exemple vous vouliez que le canal devienne modéré, vous devriez utiliser `serv.mode(canal, "+m")` . Pour qu'il ne le soit plus, ce serait `serv.mode(canal, "-m")` . Vous pouvez cumuler l'ajout et / ou le retrait de modes. Par exemple, si vous utilisez `serv.mode(canal, "+is-m")` , alors :

- le canal n'est désormais accessible que sur invitation ;
- il devient secret ;
- et il n'est plus modéré.

Viennent ensuite les modes avec paramètres.

Les principaux modes de canaux avec paramètre

Mode	Paramètre	Signification
v	Utilisateur	La personne est privilégiée.
h	Utilisateur	La personne est un *half operators* .
o	Utilisateur	La personne est un opérateur.
b	Masque de bannissement	Si une personne correspond au masque, elle ne peut pas rejoindre le canal.
e	Masque d'exception	Si une personne correspond au masque, les bannissements ne l'empêchent pas de rejoindre le canal.
k	Clé	Il faut connaître la clé pour rejoindre le canal.

Mettons que vous vouliez que bouh devienne *half operator* . La commande serait alors `serv.mode(canal, "+h bouh")` . Pour qu'il ne le soit plus, ce serait `serv.mode(canal, "-h bouh")` . Là aussi, vous pouvez combiner les modes, en séparant les paramètres par des espaces et en les mettant dans l'ordre où les modes correspondants apparaissent. Par exemple, voici ce que fait le code `serv.mode(canal, "+ohhm-v TomicBomb bouh A-dream bouletman")` en lisant de gauche à droite :

- TomicBomb devient opérateur ;
- bouh et A-dream deviennent *half operators* ;
- le canal devient modéré ;
- et bouletman perd son statut privilégié.

ircbot définit une classe pour savoir qui est présent sur un canal et quels modes y sont activés, la classe ircbot.Channel. Nous allons voir les méthodes correspondant aux modes étudiés plus haut, dans des tableaux pour changer.

Méthodes en rapport avec les utilisateurs

En français	En anglais	Liste des...	pseudonyme en est-il un ?
Utilisateurs	*Users*	`irclib.Channel.users(self)`	`irclib.Channel.has_user(self, pseudonyme)`
Privilégiés	*Voices*	`irclib.Channel.voiced(self)`	`irclib.Channel.is_voiced(self, pseudonyme)`
« Demi-opérateurs »	*Half operators*	`irclib.Channel.halfopers(self)`	`irclib.Channel.is_halfoper(self, pseudonyme)`
Opérateurs	*Operators*	`irclib.Channel.opers(self)`	`irclib.Channel.is_oper(self, pseudonyme)`

Les utilisateurs regroupent à la fois les utilisateurs lambda, les privilégiés, les *half operators* et les opérateurs. Les méthodes en rapport avec les *half operators* ne sont présentes que dans la version modifiée ; dans la version publique, celles-ci sont situées dans la liste des utilisateurs avec le préfixe %.

Méthodes pour savoir les modes activés

Le canal est-il...	irclib Channel
... modéré ?	`is_moderated(self)`
... secret ?	`is_secret(self)`
... protégé ?	`is_protected(self)`
... accessible uniquement sur invitation ?	`is_invite_only(self)`
... protégé par une clé ?	`has_key(self)`

Vous pouvez récupérer la clé du canal avec la méthode `irclib.Channel.key(self)` , qui retourne `None` s'il n'y en a pas. Pour ceux qui préfèrent utiliser les noms des modes plutôt que de longs noms de méthodes, la classe prévoit une méthode `irclib.Channel.has_mode(self, mode)` .
Maintenant que l'on a vu ce que permettait la classe, il serait bon de l'utiliser. La classe mère ircbot.SingleServerIRCBot possède un attribut channels qui est un dictionnaire faisant correspondre un canal à un objet ircbot.Channel, lequel est mis à jour automatiquement suivant les évènements.
Retournons donc voir notre *bot* de modération pour lui interdire d'exclure les membres privilégiés :

Code : Python

```
class BotModeration(ircbot.SingleServerIRCBot):
    def on_pubmsg(self, serv, ev):
        auteur = irclib.nm_to_n(ev.source())
canal = (ev.target(), self.channels[ev.target()]) # tuple (nom,
ircbot.Channel)
        message = ev.arguments()[0].lower()

if not canal[1].is_voiced(auteur) and not
canal[1].is_halfoper(auteur) and not canal[1].is_oper(auteur):
            for insulte in self.insultes:
                if insulte in message:
                    serv.kick(canal[0], auteur, "Les insultes ne
sont pas autorisées ici !")
                    break
```

Vous devriez maintenant pouvoir faire le *bot* IRC que vous souhaitez. Vous pouvez compléter ce tutoriel en lisant la documentation et / ou le code des modules étudiés, ou en vous renseignant sur le protocole IRC. Laissez libre cours à votre imagination, et amusez-vous bien !

Les images 16x16 associées aux serveurs, utilisateurs et canaux dans le tableau des évènements proviennent de http://www.famfamfam.com/.

Partager

Ce tutoriel a été corrigé par les zCorrecteurs.