pyCraft Documentation

Release 0.4.0

Ammar Askar

Jun 09, 2017

Contents

`pyCraft` is a python project to handle networking between a Minecraft server as a client.

The authentication package contains utilities to manage communicating with Mojang's authentication servers in order to log in with a minecraft account, edit profiles etc

The Connection class under the networking package handles connecting to a server, sending packets, listening for packets etc

Contents:

CHAPTER 1

Authentication

The authentication module contains functions and classes to facilitate interfacing with Mojang's Yggdrasil authentication service.

Logging In

The most common use for this module in the context of a client will be to log in to a Minecraft account. The first step to doing this is creating an instance of the AuthenticationToken class after which you may use the authenticate method with the user's username and password in order to make the AuthenticationToken valid.

class `minecraft.authentication.`**`AuthenticationToken`**(*username=None*, *access_token=None*, *client_token=None*)

Represents an authentication token.

See http://wiki.vg/Authentication.

Constructs an *AuthenticationToken* based on *access_token* and *client_token*.

Parameters: access_token - An *str* object containing the *access_token*. client_token - An *str* object containing the *client_token*.

Returns: A *AuthenticationToken* with *access_token* and *client_token* set.

`authenticate`(*username*, *password*)

Authenticates the user against https://authserver.mojang.com using *username* and *password* parameters.

Parameters:

username - An *str* object with the username (unmigrated accounts) or email address for a Mojang account.

password - An *str* object with the password.

Returns: Returns *True* if successful. Otherwise it will raise an exception.

Raises: minecraft.exceptions.YggdrasilError

Upon success, the function returns True, on failure a YggdrasilError is raised. This happens, for example if an incorrect username/password is provided or the web request failed.

exception minecraft.authentication.**YggdrasilError**(*message=None*, *status_code=None*, *yggdrasil_error=None*, *yggdrasil_message=None*, *yggdrasil_cause=None*)

Base *Exception* for the Yggdrasil authentication service.

Parameters

- **message** (*str*) – A human-readable string representation of the error.
- **status_code** (*int*) – Initial value of *status_code*.
- **yggdrasil_error** (*str*) – Initial value of *yggdrasil_error*.
- **yggdrasil_message** (*str*) – Initial value of *yggdrasil_message*.
- **yggdrasil_cause** (*str*) – Initial value of *yggdrasil_cause*.

status_code = None
int or *None*. The associated HTTP status code. May be set.

yggdrasil_cause = None
str or *None*. The *"cause"* field of the Yggdrasil response: a string containing additional information about the error. May be set.

yggdrasil_error = None
str or *None*. The *"error"* field of the Yggdrasil response: a short description such as *"Method Not Allowed"* or *"ForbiddenOperationException"*. May be set.

yggdrasil_message = None
str or *None*. The *"errorMessage"* field of the Yggdrasil response: a longer description such as *"Invalid credentials. Invalid username or password."*. May be set.

Arbitary Requests

You may make any arbitary request to the Yggdrasil service with the _make_request method passing in the AUTH_SERVER as the server parameter.

minecraft.authentication.**AUTH_SERVER = 'https://authserver.mojang.com'**
The base url for Ygdrassil requests

minecraft.authentication.**_make_request**(*server*, *endpoint*, *data*)
Fires a POST with json-packed data to the given endpoint and returns response.

Parameters: endpoint - An *str* object with the endpoint, e.g. "authenticate" data - A *dict* containing the payload data.

Returns: A *requests.Request* object.

Example Usage

An example of making an arbitary request can be seen here:

```
payload = {'username': username,
           'password': password}
```

```
authentication._make_request(authentication.AUTH_SERVER, "signout", payload)
```

CHAPTER 2

Connecting to Servers

Your primary dealings when connecting to a server will be with the Connection class

class minecraft.networking.connection.**Connection**(*address*, *port=25565*, *auth_token=None*, *username=None*, *initial_version=None*, *allowed_versions=None*, *handle_exception=None*)

This class represents a connection to a minecraft server, it handles everything from connecting, sending packets to handling default network behaviour

Sets up an instance of this object to be able to connect to a minecraft server.

The connect method needs to be called in order to actually begin the connection

Parameters

- **address** – address of the server to connect to
- **port(int)** – port of the server to connect to
- **auth_token** – *minecraft.authentication.AuthenticationToken* object. If None, no authentication is attempted and the server is assumed to be running in offline mode.
- **username** – Username string; only applicable in offline mode.
- **initial_version** – A Minecraft version string or protocol version number to use if the server's protocol version cannot be determined. (Although it is now somewhat inaccurate, this name is retained for backward compatibility.)
- **allowed_versions** – A set of versions, each being a Minecraft version string or protocol version number, restricting the versions that the client may use in connecting to the server.
- **handle_exception** – A function to be called when an exception occurs in the client's networking thread, taking 2 arguments: the exception object 'e' as in 'except Exception as e', and a 3-tuple given by sys.exc_info(); or None for the default behaviour of raising the exception from its original context; or False for no action. In any case, the networking thread

will terminate, the exception will be available via the 'exception' and 'exc_info' attributes of the 'Connection' instance.

connect()
Attempt to begin connecting to the server. May safely be called multiple times after the first, i.e. to reconnect.

disconnect()
Terminate the existing server connection, if there is one.

register_packet_listener(*method*, **args*)
Registers a listener method which will be notified when a packet of a selected type is received

Parameters

- **method** – The method which will be called back with the packet
- **args** – The packets to listen for

status(*handle_status=None*, *handle_ping=False*)
Issue a status request to the server and then disconnect.

Parameters

- **handle_status** – a function to be called with the status dictionary None for the default behaviour of printing the dictionary to standard output, or False to ignore the result.
- **handle_ping** – a function to be called with the measured latency in milliseconds, None for the default handler, which prints the latency to standard outout, or False, to prevent measurement of the latency.

write_packet(*packet*, *force=False*)
Writes a packet to the server.

If force is set to true, the method attempts to acquire the write lock and write the packet out immediately, and as such may block.

If force is false then the packet will be added to the end of the packet writing queue to be sent 'as soon as possible'

Parameters

- **packet** – The `network.packets.Packet` to write
- **force(bool)** – Specifies if the packet write should be immediate

Writing Packets

The packet class uses a lot of magic to work, here is how to use them. Look up the particular packet you need to deal with, for this example let's go with the `KeepAlivePacket`

class `minecraft.networking.packets.`**KeepAlivePacket**(*context=None*, ***kwargs*)

definition = [{'keep_alive_id': <class 'minecraft.networking.types.VarInt'>}]

Pay close attention to the definition attribute, and how our class variable corresponds to the name given from the definition:

```
packet = KeepAlivePacket()
packet.keep_alive_id = random.randint(0, 5000)
connection.write_packet(packet)
```

and just like that, the packet will be written out to the server.

It is possible to implement your own custom packets by subclassing `minecraft.networking.packets.Packet`. Read the docstrings and follow the examples in packets.py for more details on how to do advanced tasks like having a packet that is compatible across multiple protocol versions.

Listening for Certain Packets

Let's look at how to listen for certain packets, the relevant method being

`Connection.`**`register_packet_listener`**(*method*, **args*)

Registers a listener method which will be notified when a packet of a selected type is received

Parameters

- **`method`** – The method which will be called back with the packet
- **`args`** – The packets to listen for

An example of this can be found in the `start.py` headless client, it is recreated here:

```
connection = Connection(options.address, options.port, auth_token=auth_token)
connection.connect()

def print_chat(chat_packet):
    print "Position: " + str(chat_packet.position)
    print "Data: " + chat_packet.json_data

from minecraft.networking.packets import ChatMessagePacket
connection.register_packet_listener(print_chat, ChatMessagePacket)
```

The field names `position` and `json_data` are inferred by again looking at the definition attribute as before

class `minecraft.networking.packets.`**`ChatMessagePacket`**(*context=None*, ***kwargs*)

`definition` **= [{'json_data': <class 'minecraft.networking.types.String'>}, {'position': <class 'minecraft.networking.ty**

Python Module Index

m

Index

Symbols

A

C

D

K

M

R

S

W

Y