pyCraft Documentation

Release 0.6.0

Ammar Askar

Jun 08, 2019

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.

1.1 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*, *invalidate_previous=False*)

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. invalidate_previous - A *bool*. When *True*, invalidate all previously acquired 'access_token's across all clients.

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.

1.2 Arbitrary Requests

You may make any arbitrary 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.

1.2.1 Example Usage

An example of making an arbitrary 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*, *handle_exit=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** – The final exception handler. This is triggered when an exception occurs in the networking thread that is not caught normally. After any other user-registered exception handlers are run, the final exception (which may be the original exception or

one raised by another handler) is passed, regardless of whether or not it was caught by another handler, to the final handler, which may be a function obeying the protocol of 'register_exception_handler'; the value 'None', meaning that if the exception was otherwise uncaught, it is re-raised from the networking thread after closing the connection; or the value 'False', meaning that the exception is never re-raised.

- **handle_exit** – A function to be called when a connection to a server terminates, not caused by an exception, and not with the intention to automatically reconnect. Exceptions raised from this function will be handled by any matching exception handlers.

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

disconnect(*immediate=False*)
Terminate the existing server connection, if there is one. If 'immediate' is True, do not attempt to write any packets.

exception_handler(**exc_types*, ***kwds*)
Shorthand decorator to register a function as an exception handler.

listener(**packet_types*, ***kwds*)
Shorthand decorator to register a function as a packet listener.

register_exception_handler(*handler_func*, **exc_types*, ***kwds*)
Register a function to be called when an unhandled exception occurs in the networking thread.

When multiple exception handlers are registered, they act like 'except' clauses in a Python 'try' clause, with the earliest matching handler catching the exception, and any later handlers catching any uncaught exception raised from within an earlier handler.

Regardless of the presence or absence of matching handlers, any such exception will cause the connection and the networking thread to terminate, the final exception handler will be called (see the 'handle_exception' argument of the 'Connection' contructor), and the original exception - or the last exception raised by a handler - will be set as the 'exception' and 'exc_info' attributes of the 'Connection'.

Parameters **handler_func** – A function taking two arguments: the exception

object 'e' as in 'except Exception as e:', and the corresponding 3-tuple given by 'sys.exc_info()'. The return value of the function is ignored, but any exception raised in it replaces the original exception, and may be passed to later exception handlers.

Parameters **exc_types** – The types of exceptions that this handler shall

catch, as in 'except (exc_type_1, exc_type_2, ...) as e:'. If this is empty, the handler will catch all exceptions.

Parameters **early** – If 'True', the exception handler is registered before

any existing exception handlers in the handling order.

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

If `minecraft.networking.connection.IgnorePacket` is raised from within this method, no subsequent handlers will be called. If 'early=True', this has the additional effect of preventing the default in-built action; this could break the internal state of the 'Connection', so should be done with care. If, in addition, 'outgoing=True', this will prevent the packet from being written to the network.

Parameters

- **method** – The method which will be called back with the packet

- **packet_types** – The packets to listen for
- **outgoing** – If 'True', this listener will be called on outgoing packets just after they are sent to the server, rather than on incoming packets.
- **early** – If 'True', this listener will be called before any built-in default action is carried out, and before any listeners with 'early=False' are called. If 'outgoing=True', the listener will be called before the packet is written to the network, rather than afterwards.

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

2.1 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 `serverbound.play.KeepAlivePacket`

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

definition = None

classmethod field_enum(*field*, *context=None*)
The subclass of 'minecraft.networking.types.Enum' associated with this field, or None if there is no such class.

field_string(*field*)
The string representation of the value of a the given named field of this packet. Override to customise field value representation.

fields
An iterable of the names of the packet's fields, or None.

write_fields(*packet_buffer*)

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

```
from minecraft.networking.packets import serverbound
packet = serverbound.play.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 in packets.py and follow the examples in its subpackages for more details on how to do advanced tasks like having a packet that is compatible across multiple protocol versions.

2.2 Listening for Certain Packets

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

`Connection.`**`register_packet_listener`**(*method*, **packet_types*, ***kwds*)

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

If `minecraft.networking.connection.IgnorePacket` is raised from within this method, no subsequent handlers will be called. If 'early=True', this has the additional effect of preventing the default in-built action; this could break the internal state of the 'Connection', so should be done with care. If, in addition, 'outgoing=True', this will prevent the packet from being written to the network.

Parameters

- **`method`** – The method which will be called back with the packet
- **`packet_types`** – The packets to listen for
- **`outgoing`** – If 'True', this listener will be called on outgoing packets just after they are sent to the server, rather than on incoming packets.
- **`early`** – If 'True', this listener will be called before any built-in default action is carried out, and before any listeners with 'early=False' are called. If 'outgoing=True', the listener will be called before the packet is written to the network, rather than afterwards.

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.clientbound.play 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.clientbound.play.`**`ChatMessagePacket`**(*context=None*, ***kwargs*)

`class Position`

`CHAT = 0`

`GAME_INFO = 2`

SYSTEM = 1

classmethod name_from_value(*value*)

definition = [{'json_data': <class 'minecraft.networking.types.basic.String'>}, {'pos

classmethod field_enum(*field*, *context=None*)
The subclass of 'minecraft.networking.types.Enum' associated with this field, or None if there is no such class.

field_string(*field*)
The string representation of the value of a the given named field of this packet. Override to customise field value representation.

fields
An iterable of the names of the packet's fields, or None.

write_fields(*packet_buffer*)

Python Module Index

m

Index

Symbols

A

C

D

E

F

G

K

L

M

N

R

S

W

Y