

---

# Minecraft-Connection-Tools

*Release 1.3.0*

Owen Cochell

Mar 11, 2024



## CONTENTS:

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.2	Python . . . . .	3
1.3	Instillation via PIP . . . . .	4
1.4	Source Code . . . . .	4
<b>2</b>	<b>Client Tutorial</b>	<b>5</b>
2.1	Introduction . . . . .	5
2.2	Client Methods . . . . .	5
2.3	Instantiating Clients . . . . .	7
2.4	Packets . . . . .	9
2.5	Context Managers . . . . .	9
2.6	Exceptions . . . . .	9
2.7	Conclusion . . . . .	10
<b>3</b>	<b>RCON Tutorial</b>	<b>11</b>
3.1	Introduction . . . . .	11
3.2	Basic RCON Usage . . . . .	11
3.3	Conclusion . . . . .	14
<b>4</b>	<b>Query Tutorial</b>	<b>15</b>
4.1	Introduction . . . . .	15
4.2	Basic Query Usage . . . . .	15
4.3	Conclusion . . . . .	17
<b>5</b>	<b>Ping Tutorial</b>	<b>19</b>
5.1	Introduction . . . . .	19
5.2	Basic Ping usage . . . . .	19
5.3	Conclusion . . . . .	21
<b>6</b>	<b>Formatting Tutorial</b>	<b>23</b>
6.1	Introduction . . . . .	23
6.2	Note on color support . . . . .	24
6.3	Formatter Usage . . . . .	24
6.4	FormatterCollection class . . . . .	26
6.5	Custom Formatter . . . . .	29
6.6	Conclusion . . . . .	29
<b>7</b>	<b>mcli usage</b>	<b>31</b>
7.1	Introduction . . . . .	31
7.2	General Usage . . . . .	31

7.3	RCON Usage . . . . .	33
7.4	QUERY Usage . . . . .	33
7.5	PING Usage . . . . .	33
7.6	Examples . . . . .	34
7.7	Screenshots . . . . .	34
7.8	Conclusion . . . . .	36
<b>8</b>	<b>API Reference</b>	<b>37</b>
8.1	Exceptions . . . . .	37
8.2	MClient Reference . . . . .	38
8.3	Packet Reference . . . . .	45
8.4	Encoding Reference . . . . .	49
8.5	Protocol Reference . . . . .	51
8.6	Formatting Tools Reference . . . . .	54
<b>9</b>	<b>Indices and tables</b>	<b>61</b>
	<b>Python Module Index</b>	<b>63</b>
	<b>Index</b>	<b>65</b>

This is the documentation for Minecraft-Connection-Tools(mctools) - an easy to use python library for querying, interacting, and gathering information on Minecraft servers.

mctools implements the following Minecraft protocols:

1. RCON
2. Query
3. Server List Ping

Here is a brief example of the RCONClient, one of the Minecraft client implementations:

```
from mctools import RCONClient # Import the RCONClient

HOST = 'mc.server.net' # Hostname of the Minecraft server
PORT = 1234 # Port number of the RCON server

# Create the RCONClient:

rcon = RCONClient(HOST, port=PORT)

# Login to RCON:

if rcon.login("password"):

    # Send command to RCON - broadcast message to all players:

    resp = rcon.command("broadcast Hello RCON!")
```

We also offer a CLI front end for mctools, called 'mcli.py'. You can read all about it below.

This documentation contains tutorials on basic usage, as well as the API reference. We recommend starting with the client tutorial, as it gives you a general idea of how clients work.



## INSTALLATION

### 1.1 Introduction

This section of the documentation shows how to install mctools. We will cover multiple installation options here, such as installing via pip and getting the source code.

### 1.2 Python

Before you install mctools, you must first have python and pip installed. We will walk through the process of achieving this. mctools supports python 3.6 and above, but we recommend using the current latest version of python.

More information on installing/configuring python can be found [here](#)

#### 1.2.1 Linux

You can install python using your system's package manager. Below, we will install the default python3 and pip using apt, the Debian package manager:

```
$ apt install python3 python3-pip
```

It is important to specify that we want the python 3 version of pip, or else it will not work correctly.

#### 1.2.2 Windows

Windows users can download python from the [python website](#). The installation is pretty straightforward, although we recommend adding python to your PATH environment variable, as it makes using python much easier.

#### 1.2.3 Mac

You can find installation instructions [here](#).

## 1.3 Instillation via PIP

You can install mctools using PIP like so:

```
$ pip install mctools
```

You can optionally install mctools with extra color support:

```
$ pip install mctools[color]
```

Reasons on why you should install with color support are outlined in the [Formatting Tutorial](#). Color support is only relevant for Windows installations.

To learn more about PIP and installing in general, check out the [Tutorial on installing packages](#).

## 1.4 Source Code

You can acquire the source code from github like so:

```
$ git clone https://github.com/Owen-Cochell/mctools
```

This will download the source code to your computer. You can directly reference the package from your application, or install it using pip:

```
$ cd mctools # Move to the directory  
$ pip install .
```

You can also get the tarball from github, which you can download like so:

```
$ CURL -oL https://github.com/Owen-Cochell/mctools/tarball/master
```



## CLIENT TUTORIAL

### 2.1 Introduction

This tutorial will give you insight and examples into common client features. This is a general guide on client functionality. If you need specific information on a topic, like interacting with a server via RCON, see the specific tutorial for that topic.

Now, all clients inherit from the BaseClient interface. It defines features and usage present in all clients, regardless of operation. The default configuration for clients are recommended for most users, as it will do the following:

1. Generate a request ID based on system time
2. Automatically replace format codes with appropriate ASCII values
3. Set socket timeout to 60 seconds

However, advanced users may find tweaking these features to be helpful.

---

**Note:** In all following examples, we assume that a client is properly imported, and has been instantiated under the name 'client'

---

### 2.2 Client Methods

All clients have the following methods:

1. `is_connected()`
2. `start()`
3. `stop()`
4. `raw_send()`
5. `get_formatter()`
6. `set_timeout()`

### 2.2.1 is\_connected

You can use this method to determine if a client is connected:

```
value = client.is_connected()
```

This method returns a boolean, where True means we are connected, and False means we are not. 'Connected' simply means that we have established a TCP stream with the server (Or started communicating in the case of UDP), and we are ready to send/receive information.

---

**Note:** Just because we are connected doesn't mean we are authenticated! In the case of RCON, you must authenticate first before sending commands.

---

### 2.2.2 start

You can use this method to start the client, and any underlying protocol objects:

```
client.start()
```

This will, again, start the underlying protocol object, meaning that if we are communicating over TCP, a TCP stream will be established.

start() is called automatically where appropriate, so you will not have to start the object yourself.

### 2.2.3 stop

You can use this function to stop the client, and close any underlying protocol objects:

```
client.stop()
```

It is **HIGHLY** recommended to call this when you are done communicating with the server. Not doing so could cause problems server side. Protocol objects will attempt to gracefully close the connection when they are deleted, but this does not always work.

You can stop the client multiple times without any issues, and stopping a client that is already stopped will have no effect.

---

**Note:** As of mctools version 1.2.0, client objects can be started again after they have been stopped.

However, in older versions of mctools, clients can not be started again after they have been stopped. This means that if you stop a client, it will be completely unusable. This is due to the design of python sockets, as sockets can't be re-used if closed. In this case, you will have to create a new client if you stop your current one.

---

## 2.2.4 raw\_send

**Warning:** It is recommended to use the high-level wrappers, as sending your own content could mess up the client instance!

This function gives you the ability to bypass the higher-level client wrappers and send your own information:

```
client.raw_send(*args)
```

The usage of this command differs from client to client. See the documentation for specific client usage.

## 2.2.5 get\_formatters

This function gives you access to the underlying Formatters instance:

```
format = client.get_formatter()
```

This returns the FormatterCollection instance in use by the client, which will allow you to fine tune the formatter to your use.

More information can be found in the [Formatter Tutorial](#)..

## 2.2.6 set\_timeout

This function sets the timeout for network operations:

```
client.set_timeout(10)
```

The above statement will set the timeout value to 10 seconds.

## 2.3 Instantiating Clients

All clients have the same parameters when instantiating:

```
class Client(host, port=[Port Num], reqid=None, format_method='replace', timeout=60)
```

A client implementation. All clients share this format.

### Parameters

- **host** – Hostname of the server
- **port** – Port number of the server
- **reqid** – Request ID to use
- **format\_method** – Format method to use
- **timeout** – Timeout for socket operations

We can use these parameters to change the operation of clients.

### 2.3.1 host

The host of the server we are connecting to, this should be a string.

### 2.3.2 port

The port number of the server we are connecting to, this should be a integer. The default port number differs from client to client.

### 2.3.3 reqid

**Warning:** Specifying your own request ID is not recommended! Doing so could lead to unstable operation.

The request ID is what we use to identify ourselves to a server. By default, the client generates a request ID based on system time, this occurs when the value for 'reqid' is None.

You may specify your own request ID by passing an integer to the 'reqid' parameter.

### 2.3.4 format\_method

This parameter specifies how (or how not) packets should be formatted. Minecraft has a special formatting convention that allows users to add custom colors or effects to text. Info on that can be found [here](#).

Sometimes, often with the use of extensive plugins, there can be many format characters within the received data, which can make it difficult to read the content. Clients provide formatting methods to make this content more human-readable.

Clients support the following format methods, and use the following constants to identify them:

1. client.REPLACE - Replace all format characters with their appropriate ASCII values
2. client.REMOVE - Remove all format characters
3. client.RAW - Do not format the content

For example, if you wanted to remove format characters, you would instantiate the client like so:

```
client = Client('example.host', 12345, format_method=Client.REMOVE)
```

This will configure the client to remove all format characters. This logic applies to the other format options. The default operation is to replace format characters.

You can also specify the formatting operation on a per-call basis.

For example, let's say you are communicating via RCON, and want to remove the formatting characters from the 'help' command, instead of replace them. You would call the 'command' function like so:

```
resp = rcon.command('help', format_method=Client.REMOVE)
```

Every client method where 'formattable' information is fetched has a 'format\_method' parameter that you can use to set a 'one time' formatting mode. If not specified, then the global formatting type will be used.

For more information on formatters, please see the [Formatter Tutorial](#).

### 2.3.5 timeout

This parameter specifies the timeout length for socket operations. It is 60 seconds by default, but can be however long/short you want it to be. The value **MUST** be an integer. We don't recommend setting this value too high or too low.

You can change the timeout at any time using the 'set\_timeout' method. Here is an example of this in action:

```
client.set_timeout(120)
```

In this example, we have set the socket timeout to 120 seconds. All clients have the 'set\_timeout' method.

## 2.4 Packets

By default, clients only return the most relevant parts of a package, usually a payload. However, some users might want to work with the packages directly. All client methods that return server information/statistics can return the raw packets instead of the payloads. This can be done by setting the 'return\_pack' argument to 'True'.

Here is an example of this using the PINGClient:

```
pack = ping.get_stats(return_pack=True)
```

## 2.5 Context Managers

All clients have context manager support:

```
with Client('example.host', port=12345) as client:
    client.do_something()
    client.do_another_thing()
```

When the 'with' block is exited (or an exception occurs), then the stop() method will automatically be called. This ensures that the client always gracefully stops the connection.

## 2.6 Exceptions

Each client has their own set of exceptions that are raised when necessary. However, individual clients do not raise exceptions when network issues occur, which is where 'ProtocolErrors' come in.

A 'ProtocolError' is an exception raised by the underlying protocol object that each client uses. This means that it does not matter which client you are using, if a network issue occurs, then a 'ProtocolError' will be raised.

List of 'ProtocolErrors':

1. ProtocolError - Base exception for all protocol errors
2. ProtoConnectionClosed - Raised when the connection is closed by the remote host

Here is an example of importing and handling these exceptions:

```
from mctools.errors import ProtoConnectionClosed # Import the exception we wish to
↳ handle

with Client('example.host', port=1234) as client:

    try:

        client.do_something()

    except ProtoConnectionClosed:

        # Exception has been handled, and the client has been stopped:

        print("Remote host closed connection!")
```

## 2.7 Conclusion

That concludes our tutorial for client usage!

The tutorials on other topics, such as RCON, will focus on topic specific usage, and will skip generic client features.

## RCON TUTORIAL

### 3.1 Introduction

Welcome to the tutorial for RCONClient! This section aims to give you a basic understanding of RCONClient usage.

### 3.2 Basic RCON Usage

mctools gives you the ability to interact with a server via RCON. The [Minecraft RCON protocol](#) allows admins to remotely execute Minecraft commands. The Minecraft RCON protocol is an implementation of the [Source RCON protocol](#).

---

**Note:** While the RCON protocol is a common standard, we do not recommend using RCONClient for anything other than Minecraft. There has been no testing done on other RCON implementations.

---

#### 3.2.1 Creating The Instance

First, you must create a RCONClient instance:

```
from mctools import RCONClient

rcon = RCONClient('mc.server.net')
```

This will create a RCONClient instance with the default configuration. All of the options can be set as you see fit, but the default configuration is recommended for most users. RCON can only be used if the server has [RCON functionality enabled](#).

---

**Note:** For more information on general client configuration and instantiation, see the [client tutorial](#).

---

### 3.2.2 Authenticating with the RCON server

Before you can start sending commands, you must first authenticate like so:

```
success = rcon.login('password')
```

Where 'password' is the password used to authenticate.

The function will return True for success, and False for failure. Most servers will require you to specify a password before you can send commands.

---

**Note:** By default, RCONClient will raise an exception if you attempt to send commands when not authenticated. You can read about disabling this feature below, but be aware that sending commands while not authenticated can lead to unstable operation.

---

If you have failed to authenticate, you may try again. mctools will allow you to attempt to authenticate as many times as you need, but the RCON server may have security features in place to block repeated login attempts.

If you have successfully authenticated, you can now start sending commands to the RCON server.

You can check if you are authenticated like so:

```
auth = rcon.is_authenticated()
```

This function will return True if you are authenticated, False if not.

### 3.2.3 Interacting with the RCON server

You may now send Minecraft commands to the server like so:

```
response = rcon.command("command")
```

Where 'command' is the command you wish to send to the RCON server. The function will return the response in string format from the server, formatted appropriately.

---

**Note:** You can disable the authentication check by passing 'no\_auth=True' to the command function, which will disable the check for this command. Be aware that if the server refuses to serve you, then a RCONAuthenticationError exception will be raised.

---

For example, if you wanted to broadcast 'Hello RCON!' to every player on the server, you would issue the following:

```
response = rcon.command("broadcast Hello RCON!")
```

The command sent will broadcast "Hello RCON!" to every player on the server.

---

**Note:** Sometimes, the server will respond with an empty string. Some commands have no output, or return an empty string when issued over RCON, so this is usually a normal operation. It can also mean that the server doesn't understand the command issued.

---



## RCON Outgoing Packet length

The RCON Protocol has an outgoing(client to server) packet size limitation of 1460 bytes. Taking into account the mandatory information we have to send(request ID, type, padding, ect.), the maximum command size that can be sent is 1446 bytes.

This limitation unfortunately has no workaround, and is an issue with the RCON protocol, and therefore beyond our control. mctools does implement a length check to make sure outgoing packets are not too big.

If an outgoing packet is too big, and the length check is enabled, then an 'RCONLengthError' exception will be raised, and the packet will not be sent. This ensures that any nasty side effects of going over the outgoing limit will be avoided, thus keeping the connection in a stable state.

You can optionally disable the outgoing length check by passing 'length\_check=False' to the command method.

**Warning:** Disabling outgoing length checks is not recommended! Doing so could mess up the state of your client!

Here is an example of disabling outgoing length checks:

```
# Lets send a HUGE command:
# (Assume 'huge_command' is a string containing a command larger than 1446 bytes)

resp = rcon.command(huge_command, length_check=False)
```

This will prevent the 'RCONLengthError' exception from being raised, and mctools will send the large packet normally.

If a large packet is sent to the RCON server, then some nasty things could occur. The most likely is that the server will forcefully close the connection, although other unsavory events could occur. This is why we recommend keeping the outgoing length check enabled.

## RCON Incoming Packet Fragmentation

Sometimes, the RCON server will send fragmented packets. This is because RCON has an incoming(server to client) maximum packet size of 4096 bytes. RCONClient will automatically handle incoming packet fragmentation for you.

If the incoming packet is 4096 bytes in length, then we will assume the packet is fragmented. If this is the case, then mctools sends a junk packet to the server, and reads packets until the server acknowledges the junk packet. The RCON protocol ensures that all packets are sent in the order that they are received, meaning that once the server responds to the junk packet, then we can be sure that we have all of the relevant packets. We then concatenate the packets we received, and return it as one.

However, you can disable the check by passing 'frag\_check=False' to the command method.

**Warning:** Disabling fragmentation checks is not recommended! Doing so could mess up the state of your client!

Here is an example of disabling RCON packet fragmentation:

```
# Lets run a command that generates fragmentation:

resp = rcon.command("help", frag_check=False)
```

This will return the content of the first 4096 bytes. Any subsequent call to 'command' or 'raw\_send' will return the rest of the fragmented packets. This means that you will have incomplete content, and subsequent calls will return irrelevant information. Unless you have a reason for this, it is recommended to keep packet fragmentation enabled.

### 3.2.4 Ending the session

To end the session with the server correctly, do the following:

```
rcon.stop()
```

This will stop the underlying TCP connection to the server. It is ALWAYS recommended to stop the instance, as not doing so could cause problems server-side.

## 3.3 Conclusion

And that concludes the basic usage for RCONClient!

## QUERY TUTORIAL

### 4.1 Introduction

Welcome to the tutorial for QUERYClient! This section aims to give you a basic understanding on QUERYClient usage.

### 4.2 Basic Query Usage

mctools gives you the ability to retrieve server statistics via the [Minecraft Query protocol](#). The Minecraft Query protocol is an implementation of the [Gamespy Query protocol](#).

---

**Note:** While the Minecraft Query protocol was made to be compatible with the Gamespy Query protocol, we don't recommend using QUERYClient for anything other than Minecraft. No testing has been done on other Query implementations.

---

#### 4.2.1 Creating the Instance

First, you must create a QUERYClient instance:

```
from mctools import QUERYClient

query = QUERYClient('mc.server.net')
```

This will create a QUERYClient instance with the default configuration. All of the options can be set as you see fit, but the default configuration is recommended for most users. Query can only be used if the server has [Query functionality enabled](#).

---

**Note:** For more information on general client configuration and instantiation, see the [client tutorial](#).

---

## 4.2.2 Retrieving Statistics

There are two types of information you can retrieve from a server: basic stats and full stats. Basic stats show host IP, port number, message of the day, etc. This is fine for most uses. Full stats shows more information about the server, such as players connected, plugins installed, etc. We will walk through the process of retrieving both.

### 4.2.3 Basic statistics

You can retrieve basic stats like so:

```
stats = query.get_basic_stats()
```

This will fetch basic server statistics and put it into a dictionary. The dictionary has the following format:

```
{'gametype': 'SMP',  
'hostip': '127.0.0.1',  
'hostport': '25565',  
'map': 'world',  
'maxplayers': '20',  
'motd': 'Now we got business!',  
'numplayers': '1'}
```

The *gametype* field is hardcoded to 'SMP'.

The *hostip* field is the host IP of the server.

The *hostport* field is the hostport of the server.

The *map* field is the name of the current map.

The *maxplayers* field is the maximum number of players.

The *motd* field is the message of the day.

The *numplayers* are the players connected to the server.

### 4.2.4 Full statistics

If you want to retrieve the full stats, you may do the following:

```
stats = query.get_full_stats()
```

This will fetch full server statistics and put it into a dictionary. The dictionary should have the following format:

```
{'game_id': 'MINECRAFT',  
'gametype': 'SMP',  
'hostip': '127.0.0.1',  
'hostport': '25565',  
'map': 'world',  
'maxplayers': '20',  
'motd': 'Now we got business!',  
'numplayers': '1',  
'players': ['MinecraftPlayer'],  
'plugins': '',  
'version': '1.15.2'}
```

The *game\_id* field is hardcoded to 'MINECRAFT'.

The *gametype* field is hardcoded to 'SMP'.

The *hostip* field is the host IP of the server.

The *hostport* field is the port number of the server.

The *map* field is the name of current map.

The *maxplayers* field is the maximum amount of players allowed on the server.

The *motd* is the message of the day.

The *numplayers* are the number of players currently connected.

The *players* is a list containing the user names of all connected players.

The *plugins* field contains a list of plugins used. This is not used by the vanilla server, however community servers such as Bukkit and Spigot use this value. Most servers follow this format:

```
[SERVER_MOD_NAME[: PLUGIN_NAME(; PLUGIN_NAME...)]]
```

The *version* field is the version of the server.

### 4.2.5 Stopping the instance

Due to the [UDP protocol](#)'s design(The protocol Query uses), the client instance does not *need* to be stopped. However, we still recommend stopping your client for readability, and so you can be explicit as to when your program will stop communicating over the network.

## 4.3 Conclusion

That concludes the tutorial on QUERYClient usage!



## PING TUTORIAL

### 5.1 Introduction

Welcome to the tutorial for PINGClient! This section aims to give you a basic understanding on PINGClient usage.

### 5.2 Basic Ping usage

mctools gives you the ability to ping and retrieve basic statistics from a server via the [Server List Ping Interface](#). mctools uses a method similar to how the official Minecraft client pings servers. You can retrieve a lot of useful information, such as ping latency, server version, message of the day, players connected, ect.

Plus, Server List Ping is built into the protocol, meaning that it is always enabled, regardless of server configuration. This gives you a reliable way to get basic server statistics, even if RCON or Query is disabled.

**Warning:** As of now, PINGClient only supports Minecraft servers that are version 1.7 or greater.

#### 5.2.1 Creating the instance

First, you must create a PINGClient instance:

```
from mctools import PINGClient

ping = PINGClient('mc.server.net')
```

This will create a PINGClient instance with the default configuration. All of the options can be set as you see fit, but the default configuration is recommended for most users.

Unlike other clients, PINGClient allows you to optionally specify the protocol number to use when communicating. You can do this like so:

```
from mctools import PINGClient

ping = PINGClient('mc.server.net', proto_num=PROTOCOL_NUMBER)
```

This allows PINGClient to emulate certain versions of the Minecraft client. By default, we use protocol number 0, which means we are inquiring on which protocol version we should use. You can find a list of protocol numbers and their versions [here](#).

---

**Note:** For more information on general client configuration and instantiation, see the [client tutorial](#).

---

## 5.2.2 Pinging the server

**Warning:** The server will automatically close the connection after you ping the server or fetch statistics.

If you are running mctools version 1.2.0 or above, the PINGClient will be automatically stopped for you after each operation. The PINGClient can then be started again and used as expected.

However, older versions of mctools do not support client restarting, and you will have to create a new PINGClient instance.

Once you have created your PINGClient, you can ping the server:

```
elapsed = ping.ping()
```

This will return the latency for the ping in milliseconds. You can use this to determine if a server is receiving connections.

## 5.2.3 Retrieving Statistics

You can also receive statistics from the server:

```
stats = ping.get_stats()
```

This will fetch ping stats, and put it into a dictionary. The contents of the dictionary *should* be in the following format:

```
{'description':
  {'text': 'Now we got business!'},
'players':
  {'max': 20,
   'online': 1,
   'sample': [
     {'id': 'fbf11fd0-5b74-490c-adc4-91febe9de2ae',
      'name': 'MinecraftPlayer'}]},
   'message': ''},
'time': 0.09879999993245292,
'version': {
  'name': '1.15.2',
  'protocol': 578},
'favicon': 'data:image/png;base64,<data>'}
```

The *description* field is the message of the day.

The *players* field gives some information about connected players. It tells us the maximum amount of players allowed on the server at once(*max*), as well as how many players are currently connected(*online*). It also supplies a sample list of players who are online(*sample*). Some very large scale servers might not offer a sample list of connected players, and simply leave it blank.

The *message* field contains the message embedded in the player sample list. If you have formatting enabled, PINGClient will automatically separate the message and the valid players. We touch on this more later in the document.



The *time* field is the latency in milliseconds.

The *version* field gives some information about the server version. The *name* field usually contains the server version, and this can differ if the server is using a different implementation (Such as [PaperMC](#), [Spigot](#), or [Bukkit](#)). The *protocol* is the protocol number the server is using.

The *favicon* field is a [PNG](#) image encoded in [Base64](#). This field is optional, and may not be present.

### 5.2.4 Note on Packet Format

For most cases, the information received will match the example above, and each field will contain the expected values that they *should* contain.

However, some servers take it upon themselves to embed messages into the player sample list, or give the description in [ChatObject](#) notation. If you have formatting enabled, then these cases are automatically handled for you.

You can read more about the ping formatters and how they handle data in the [Formatting tutorial](#).

### 5.2.5 Stopping the instance

As of mctools version 1.2.0, the `PINGClient` is automatically stopped for you after each operation.

Still, it is recommended to stop the client anyway when it is done being used:

```
ping.stop()
```

This will stop the underlying TCP connection to the Minecraft server, if there still is a connection. Again, most of the time it is not necessary to stop the client as it is done for you. You should still do so, as it can ensure that the client is stopped in case it did not automatically stop itself. It also helps readability, and allows you to explicitly state when you are done communicating over a network.

## 5.3 Conclusion

That concludes the tutorial for `PINGClient`!



## FORMATTING TUTORIAL

### 6.1 Introduction

This section aims to give you a basic understanding of formatters and how to use them.

Minecraft servers(vanilla and otherwise) use [special formatting codes](#) for adding color and effects to text. Minecraft uses the '\$' character and the value after that to determine what color/effect the text should have.

For example:

```
§4This text will appear red.
```

When the formatting operation is complete, the text will appear red.

These colors are used in a variety of things, such as signs, books, player names, command output, ect. However, these characters can make it difficult to read the content of the message.

For example, take the following help output:

```
§e----- §fHelp: Index (1/40) §e-----  
§7Use /help [n] to get page n of help.  
§6Aliases: §fLists command aliases  
§6Bukkit: §fAll commands for Bukkit  
§6ClearLag: §fAll commands for ClearLag  
§6Essentials: §fAll commands for Essentials  
§6LuckPerms: §fAll commands for LuckPerms  
§6Minecraft: §fAll commands for Minecraft  
§6Vault: §fAll commands for Vault  
§6WorldEdit: §fAll commands for WorldEdit
```

This output is somewhat difficult to read. It would be easier if the formatting chars were replaced with ASCII color codes so we can see the response in color. It would also be easier if the characters were removed. mctools provides methods that can handle these formatting characters, so text can be made more human-readable.

## 6.2 Note on color support

The mctool's formatters add color and text attributes to text using [ASCII escape codes](#) and [Ascii color codes](#). In most cases, these color codes will be universally compatible, and the terminal in use will handle these codes, and draw the text with the appropriate colors/attributes.

However, some terminals will NOT support handling of ASCII escape codes, with [Windows 10 CMD.exe](#) being one of them. This can lead to output that is either not colored or difficult to read, as these terminals often output the characters instead of applying the attributes to the text.

To handle this, mctools offers an optional install dependency which will install the python library [colorama](#). colorama will automatically enable ASCII escape code support, which will allow us to draw text in certain colors.

colorama is automatically enabled if it is installed. If not installed, the formatter will continue the formatting operation.

## 6.3 Formatter Usage

A 'Formatter' is a class that alters content received from a server. All formatters must inherit the 'BaseFormatter' class, and must provide the following methods:

1. `format()` - Change the makeup of the content(insert/move characters) to make text more readable.
2. `clean()` - Remove content to make it more readable

This design was mostly for handling formatting characters, but a formatter can make any changes to the text as it sees fit.

---

**Note:** A formatter can optionally specify a 'get\_id()' method that returns an integer. This integer is used to determine the order of the formatter. The lower the number, the higher it's priority.

---

For example, lets say that a server is replacing content with an '@' symbol every few characters, and you want to make a formatter that would handle this oddity. Your `clean()` method would simply remove each and every '@' character from the content. Your `format()` method, however, would attempt to replace the '@' characters with their intended values. Again, this is just a recommendation. Both methods can do the same operation if that's what the situation entails.

mctools has some formatters built in for convenience. The builtins ONLY handle formatting characters(With the exception of the special formatters). Builtin formatters all use static methods, meaning that a class does not have to be instantiated to be used.

### 6.3.1 BaseFormatter

Parent class that each formatter MUST inherit. This class also has an ID of 20, meaning that any formatter that hasn't specified a `get_id()` method will have an ID of 20.

### 6.3.2 DefaultFormatter

DefaultFormatter is the backbone of all built in formatters. DefaultFormatter handles formatting characters, replacing them with the desired ASCII color codes, or simply removing them. DefaultFormatter only handles strings, and is used by the RCONClient, as well as other formatters.

### 6.3.3 QUERYFormatter

QUERYFormatter was designed to format a query response in dictionary format, processing the relevant fields only. It uses DefaultFormatter for replacing/removing format chars, and is used by QUERYClient.

### 6.3.4 PINGFormatter

PINGFormatter is similar to Query, because it only formats relevant fields.

However, PINGFormatter uses two other formatters, ChatObjectFormatter and SampleDescriptionFormatter to handle some scenarios that may come up during formatting process.

These formatters will do the same operation regardless of whether we are cleaning or formatting, with the exception of color codes, which will only be included if we are formatting the text.

If you set the format method to RAW, then the special formatters will not be ran, meaning that the makeup of the server statistics will not be altered.

### ChatObjectFormatter

Some newer servers send description data in [ChatObject notation](#), which uses a collection of dictionaries to format text instead of formatting characters.

Here is an example of a description in ChatObject notation:

```
{'description':
  {'extra': [{ 'bold': True,
               'color': 'yellow',
               'text': 'This is bold and yellow!' },
             { 'color': 'gold',
               'text': ' Just gold. New line!\n' },
             { 'color': 'white',
               'italics': True,
               'text': 'We are on a new line, ' },
             { 'color': 'green',
               'text': 'and we love the color green.' },
             { 'color': 'white',
               'text': '.' } ] },
  'text': '' }
```

As you can see, this makes reading and parsing the content difficult. ChatObjectFormatter fixes this problem by converting the dictionary into a single string, which makes reading and parsing the data much easier.

## SampleDescriptionFormatter

Sometimes, servers like to embed descriptions into the sample player list. Servers usually use a player with a null UUID to show message content, so this formatter attempts to separate valid players from message content.

Have a look at this example sample player list:

```
{'players': {'max': 5000,
  'online': 723,
  'sample': [{ 'id': '00000000-0000-0000-0000-000000000000',
    'name': 'We are a server.'},
    { 'id': '00000000-0000-0000-0000-000000000000',
    'name': ''},
    { 'id': '00000000-0000-0000-0000-000000000000',
    'name': 'Check out our Twitter!'},
    { 'id': '00000000-0000-0000-0000-000000000000',
    'name': ''},
    { 'id': '00000000-0000-0000-0000-000000000000',
    'name': 'We have really great players!'},
    { 'id': '00000000-0000-0000-0000-000000000000',
    'name': ''},
    { 'id': '00000000-0000-0000-0000-000000000000',
    'name': 'Here is one of them:'},
    { 'id': '2ef8ad56-ec35-46e7-b90c-8172386d3fe7',
    'name': 'MinecraftPlayer1'}]}}
```

If you look, there is a message encoded in this sample list, with one valid player. The message has a null UUID, which is how SampleDescriptionFormatter determines if a user list is actually a message.

After the formatting operation is complete, the *player* sub-dictionary will look like this:

```
{'players': {'max': 5000,
  'online': 723,
  'sample': [{ 'id': '2ef8ad56-ec35-46e7-b90c-8172386d3fe7',
    'name': 'MinecraftPlayer1'}],
  'message': 'We are a server.\n\nCheck out our Twitter!\n\nWe have really great_
→players!\n\nHere is one of them:'}}
```

Now, the sample only contains valid players, and the message is stored under a separate key named 'message'. This allows us to accurately determine who is really playing, and view the message with no extra processing.

## 6.4 FormatterCollection class

'FormatterCollection' is a class that handles a collection of formatters. It offers an easy to use API for adding/removing formatters, altering text with multiple formatters, and defining what formatter should be used for specific inputs.

Every client has a FormatterCollection instance that they use to format incoming data (Clients automatically load the relevant formatters at the start of the instance). However, clients give you the option to work with formatters directly (This can be done by calling the 'get\_formatter()' method of the class).

FormatterCollection offers the following methods to work with:

1. add(formatter, command, ignore=None, args\*, kwargs\*) - Add a formatter
2. remove(formatter) - Remove a formatter

3. `clear()` - Remove all formatters
4. `format(text, command)` - Formats the given text
5. `clean(text, command)` - Cleans the given text
6. `get()` - Returns the list of formatters

We will go over each of these methods and their usage.

### 6.4.1 add

The 'add' method adds a formatter to the collection. It has the following parameters:

1. `formatter` - Formatter class to add (Must inherit `BaseFormatter`, or an exception will be raised)
2. `command` - Command(s) associated with the formatter
3. `ignore` - Command(s) to ignore

**Warning:** `FormatterCollection` will NOT instantiate your formatters. This means you must instantiate your formatter BEFORE adding it to the collection, or make all of your formatter methods static.

`FormatterCollection` will only call a formatter that can handle relevant text. To determine if text is relevant to a formatter, you must supply command(s) to the add method. The value can be a string containing a single command, or a list containing multiple. You may also supply a empty string('') to the command parameter to affiliate the formatter to every command (unless a command is ignored, which we will get to later).

For example, let's say you wanted to create a formatter for `RCONClient` that only handles output from the 'motd' command. You would add the formatter like this (assume that `FormatterCollection` is instantiated as 'form').

```
form.add(my_formatter, 'motd')
```

Content will only be sent to this formatter if the command executed was 'motd'. Clients automatically supply the command executed when formatting content.

Conversely, we have the 'ignore' parameter. Ignore specifies which commands should be ignored by the formatter. Unlike the command parameter, ignore is optional, and nothing will be ignored if it is not set. ignore accepts commands in the same format as the command parameter, meaning that you can specify a single command as a string, or multiple commands as a list.

For example, if you want to add a formatter that accepts all commands except the 'list' command, you can do the following:

```
form.add(my_formatter, '', ignore='list')
```

`FormatterCollection` will send all content to the formatter, except content from the 'list' command.

As you can see, the convention for affiliating commands with a formatter is primarily designed for usage with `rcon`. However, `FormatterCollection` also supplies the following constants to help identify formatting operations:

1. `QUERY` - Value used by `QUERYClient` to identify query content to be formatted
2. `PING` - Value used by `PINGClient` to identify ping content to be formatted

You can use these values to affiliate or ignore ping/query content. For example, let's say you want to make a formatter that handles ping content. You can do the following:

```
form.add(my_formatter, form.PING)
```

This will affiliate the formatter with ping content.

### 6.4.2 remove

Will remove a specified formatter from the collection. You must supply the formatter instance to be removed. Will return True for success, False for failure.

### 6.4.3 clear

Will remove all formatters from the collection.

### 6.4.4 format

Will send the given content through the relevant formatters, and format the content. You must supply the command issued with the 'command' parameter. The same logic from above applies here, if you pass '', then it will affiliate the content with ALL formatters.

### 6.4.5 clean

Same operation, except the the content is cleaned by relevant formatters instead of formatted.

### 6.4.6 get

**Warning:** It is not recommended to edit this list yourself! Users should use the higher level methods to add or remove formatters.

This will return the list of formatters used by FormatterCollection. The information on a formatter is stored in a sublist with the following format:

```
[Formatter Instance,  
Commands to match,  
Commands to ignore]
```

You can edit this list manually, however it is recommended that you use the higher level methods for adding/removing formatters.



## 6.5 Custom Formatter

In this example we will be writing a formatter that will replace the word 'help' with 'assistance'. If we are cleaning the text, then we will simply remove 'help' from the content.

```
from mctools import formattertools

class HelpFormatter(formattertools.BaseFormatter)
    """
    A Simple formatter to change some wording on the Minecraft help menu.
    """

    @staticmethod
    def format(text):
        """
        Replaces 'help' with assistance.
        """

        return text.replace('help', 'assistance')

    @staticmethod
    def clean(text):
        """
        Removes 'help' from the text.
        """

        return text.remove('help')
```

We now have a formatter we can use. We must register it with the RCONClient FormatterCollection:

```
from mctools import mclient

rcon = mclient.RCONClient('mc.server.net', 25565)

form = rcon.get_formatter()

form.add(HelpFormatter, 'help')

rcon.start()
```

The formatter has been registered with the FormatterCollection of the RCONClient, and will format the help menu accordingly.

## 6.6 Conclusion

You should now have an understanding on the usage of formatters. Most of the time, the built in formatters will handle the formatting correctly. If this is not the case, then you can create and add your own formatter to the client.



## MCLI USAGE

### 7.1 Introduction

'mcli.py' is a command line application that provides a front end for the mctools library. mcli supports all operations offered by mctools in a simple, robust manner.

### 7.2 General Usage

This section defines general usage for mcli.py

---

**Note:** From this point onwards, we reference mcli.py as mcli, which is how you would call the command if you installed via pip or setuptools.

However, if you downloaded the source code or the source distribution without installing, the you will call mcli as a python file:

```
python3 mcli.py [arguments]
```

---

#### 7.2.1 General Arguments

**usage:** mcli.py [-h] [-o PATH] [-oc | -or] [-nc] [-r] [-t TIMEOUT] [-ri REQID]  
[-q] host {rcon,query,ping}

Shows the help menu:

-h, -help

Disables color output, and removes format characters from the server output:

-nc, -no-color

Shows format characters, does not replace or remove them:

-r, -raw

Sets a timeout value for socket operations in seconds(default value is 60):

-t [TIMEOUT], -timeout [TIMEOUT]

Sets a custom request ID, instead of generating one:

-ri [REQID], -reqid [REQID]

Quiet mode, will not output anything to the terminal:

-q, --quiet

### 7.2.2 Output Options

This section defines arguments for outputting server responses to a external file.

Saves the output to a file in JSON format (only saves relevant content, such as server responses and errors):

-o [PATH], --output [PATH]

Replaces format characters with ascii color codes in the output file. Default action is to remove them:

-oc, --output-color

Leaves formatting characters when outputting to a file. Default action is to remove them:

-or, --output-raw

### 7.2.3 Hostname

After you have specified your optional arguments, you must provide a hostname to connect to. A hostname can be a domain name or IP address, anything that your computer can resolve.

If you need to use a custom port, you can specify it using the ':' character after the hostname. If no port number is specified, then the default port number for the connection you selected will be used.

For example, let's say you wanted to connect to 'mc.server.net' on port 12345:

```
mcli mc.server.net:12345 [...]
```

This will set the port number to 12345.

### 7.2.4 Specifying a Connection type

After you have specified your optional arguments and hostname, you must specify a connection type. You may choose from the following:

1. rcon
2. query
3. ping

Each connection type has specific arguments, which we will be going into. For now, an example:

```
mcli mc.server.net rcon [...]
```

This will start a RCON connection to 'mc.server.net'.

## 7.3 RCON Usage

Setting 'rcon' as the connection type will start a RCON connection to the server. The following arguments are available for RCON:

Send a command to the RCON server. Can specify multiple:

-c [COMMAND], --command [COMMAND]

Starts an interactive session with the RCON server:

-i, --interactive

After you have specified your optional parameters, you must provide a password. This is a required field.

```
mcli [OPTIONAL ARGUMENTS] mc.server.net rcon [OPTIONAL ARGUMENTS] [PASSWORD]
```

For example, let's say you wanted to start an interactive RCON session with 'mc.server.net' with the password 'Minecraft is Cool!':

```
mcli mc.server.net rcon --interactive 'Minecraft is Cool!'
```

This will create an interactive RCON session with 'mc.server.net'.

## 7.4 QUERY Usage

Setting 'query' as the connection type will start a QUERY connection to the server. The following arguments are available for QUERY:

Retrieve full stats(mcli retrieves basic stats by default):

-fs, --full-stats

For example, lets say you wanted to retrieve full Query statistics from 'mc.server.net' on port 1234:

```
mcli mc.server.net:1234 query -fs
```

## 7.5 PING Usage

Setting 'ping' as the connection type will start a PING connection to the server. The following arguments are available for PING:

Output favicon data to the terminal(mcli does not output favicon data by default):

-sf, --show-favicon

Use a custom protocol number:

-p PROTOCOL\_NUMBER

For example, lets say you wanted to ping 'mc.server.net', but pretend to be Minecraft version 1.13:

```
mcli mc.server.net ping -p 393
```

## 7.6 Examples

Below are some usage examples for mcli:

Ping server and get basic stats:

```
mcli [hostname] ping
```

Backup and stop a Minecraft server via RCON:

```
mcli [hostname] rcon --command backup --command stop [password]
```

Message player 'ILoveCraft' on 'mc.server.net' with password 'craft':

```
mcli mc.server.net rcon --command 'msg ILoveCraft Minecraft loves you too!' craft
```

Start an interactive RCON session with 'mc.example.com' on port 858585, with test as the password:

```
mcli mc.example.com:858585 rcon --interactive test
```

Get full stats via Query and output the result to 'query.txt':

```
mcli -o query.txt [hostname] query --full-stats
```

Get basic stats via query and disable color:

```
mcli --no-color [hostname] query
```

Ping server, but leave format chars:

```
mcli --raw [hostname] ping
```

## 7.7 Screenshots

Here are some screenshots of mcli in action:

### 7.7.1 RCON

Interactive RCON session

```

+=====+
# Starting TCP connection to RCON server @ 127.0.0.1:40004 ...
# Started TCP connection to RCON server @ 127.0.0.1:40004!
# Authenticating with RCON server ...
# Authentication with RCON server successful!
# Running user commands ...
+=====+

```

Welcome to the RCON interactive session!

Connection Info:

Host: 127.0.0.1

Port: 40004

Type 'q' to quit this session.

Type 'help' or '?' for info on commands!

rcon@127.0.0.1:40004>>?

----- Help: Index (1/40) -----

Use /help [n] to get page n of help.

Aliases: Lists command aliases

Bukkit: All commands for Bukkit

ClearLag: All commands for ClearLag

Essentials: All commands for Essentials

LuckPerms: All commands for LuckPerms

Minecraft: All commands for Minecraft

Vault: All commands for Vault

WorldEdit: All commands for WorldEdit

rcon@127.0.0.1:40004>>

## 7.7.2 Query

Fetching full statistics via Query

```

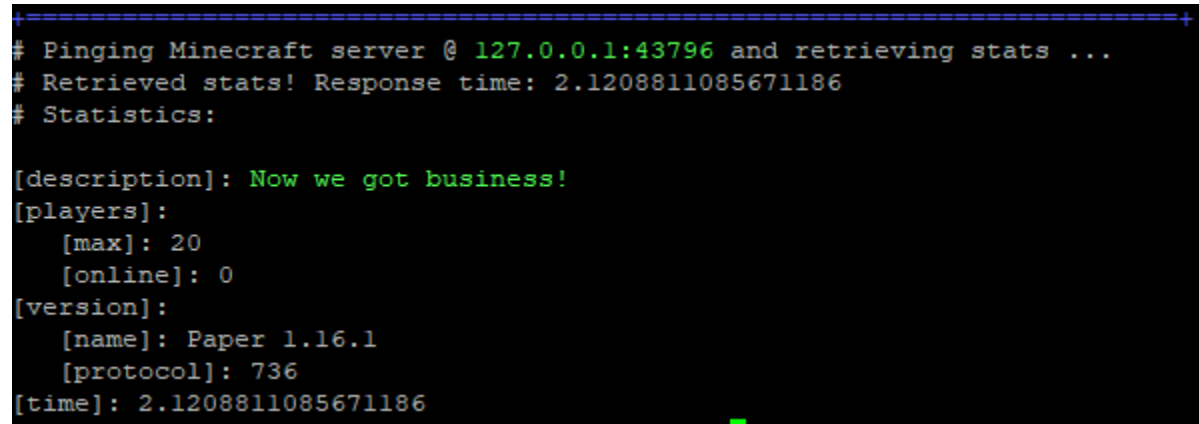
+=====+
# Authenticating with Query server @ 127.0.0.1:40003 and retrieving full stats ...
# Retrieved full stats!:

[motd]: Now we got business!
[gametype]: SMP
[game_id]: MINECRAFT
[version]: 1.16.1
[plugins]: Paper on Bukkit 1.16.1-R0.1-SNAPSHOT: ClearLag 3.1.6; LuckPerms 4.4.30;
[map]: world
[numplayers]: 0
[maxplayers]: 20
[hostport]: 43796
[hostip]: 127.0.0.1
[players]:
[0]:

```

### 7.7.3 Ping

Pinging Minecraft server and fetching statistics

A terminal window with a black background and green text. The output shows the command being executed, the response time, and a JSON-formatted list of statistics including description, players, version, and time.

```
# Pinging Minecraft server @ 127.0.0.1:43796 and retrieving stats ...
# Retrieved stats! Response time: 2.1208811085671186
# Statistics:

[description]: Now we got business!
[players]:
  [max]: 20
  [online]: 0
[version]:
  [name]: Paper 1.16.1
  [protocol]: 736
[time]: 2.1208811085671186
```

## 7.8 Conclusion

You should now have a basic understanding of the ‘mcli.py’ frontend and how to use it. After you install mctools through pip(or some other method), then the ‘mcli’ command should be available.



## API REFERENCE

Below is the API reference for mctools. Each feature is broken up into a different section. The average user shouldn't have to worry about these, and should instead use the higher level client classes and functions. You can find tutorials on how to use them elsewhere in this documentation.

### 8.1 Exceptions

Exception objects for mctools.

**exception** `mctools.errors.MCToolsError`

Base class for mctools exceptions.

**exception** `mctools.errors.PINGError`

Base class for PING errors

**exception** `mctools.errors.PINGMalformedPacketError(message)`

Exception raised if the ping packet received is malformed/broken, or not what we were expecting.

**exception** `mctools.errors.ProtoConnectionClosed(message)`

Exception raised when the connection is closed by the socket we are connected to.

This occurs when we receive empty bytes from 'recv()', as this means that the remote socket is done writing, meaning that our connection is closed.

**exception** `mctools.errors.ProtocolError`

Base class for protocol errors.

A protocol error is raised when an issue arises with the python socket object. These exceptions will be raised for ALL clients, as they all use the same backend.

**exception** `mctools.errors.RCONAuthenticationError(message)`

Exception raised when user is not authenticated to the RCON server. This is raised when a user tries to send a RCON command, and the server refuses to communicate.

#### Parameters

- **message** (*str*) – Explanation of the error
- **server\_id** – ID given to us by the server

**exception** `mctools.errors.RCONCommunicationError(message)`

Exception raised if the client has trouble communicating with the RCON server. For example, if we don't receive any information when we want a packet.

**Parameters**

**message** (*str*) – Explanation of the error

**exception** `mctools.errors.RCONError`

Base class for RCON errors.

**exception** `mctools.errors.RCONLengthError`(*message*, *length*)

Exception raised if the client attempts to send a packet that is too big, greater than length 1460.

**exception** `mctools.errors.RCONMalformedPacketError`(*message*)

Exception raised if the packet we received is malformed/broken, or not what we were expecting.

**Parameters**

**message** (*str*) – Explanation of the error

## 8.2 MClient Reference

Main Minecraft Connection Clients - Easy to use API for the underlying modules Combines the following: - Protocol Implementations - Packet implementations - Formatting tools

**class** `mctools.mclient.BaseClient`

Parent class for Minecraft Client implementations. This class has the following formatting constants, which every client inherits:

- `BaseClient.RAW` - Tells the client to not format any content.
- `BaseClient.REPLACE` - Tells the client to replace format characters.
- `BaseClient.REMOVE` - Tells the client to remove format characters.
- `BaseClient.DEFAULT` - Chooses the global default

You can use these constants in the 'format\_method' parameter in each client.

Changed in version 1.3.0.

Removed interface method 'raw\_send()', as no generic definition exists.

**gen\_reqid()** → `int`

Generates a request ID using system time.

**Returns**

System time as an integer

**Return type**

`int`

**get\_formatter()** → *FormatterCollection*

Returns the formatter used by this instance, Allows the user to change formatter operations.

**Returns**

FormatterCollection used by the client.

**Return type**

*FormatterCollection*

**is\_connected()** → `bool`

Determine if we are connected to the remote entity, whatever that may be. This raises a 'NotImplemented-Error' exception, as this function should be overridden in the child class.

**Returns**

True if connected, False if not.

**Raises**

NotImplementedError: If function is called.

**set\_timeout**(*timeout: int*)

Sets the timeout for the underlying socket object.

**Parameters**

**timeout** (*int*) – Value in seconds to set the timeout to

**start**()

Starts the client, and any protocol objects/formatters in use. (Good idea to put a call to Protocol.start() here). This raises a 'NotImplementedError' exception, as this function should be overridden in the child class.

**Raises**

NotImplementedError: If function is called.

**stop**()

Stops the client, and any protocol objects/formatters in use. (Again, good idea to put a call to Protocol.stop() here). This raises a 'NotImplementedError' exception, as this function should be overridden in the child class.

**Raises**

NotImplementedError: If function is called.

**class** mctools.mclient.**PINGClient**(*host: str, port: int = 25565, reqid: int = -1, format\_method: int = 1, timeout: int = 60, proto\_num: int = 0*)

Ping client, allows for user to interact with the Minecraft server via the Server List Ping protocol.

**Parameters**

- **host** (*str*) – Hostname of the Minecraft server
- **port** (*int*) – Port of the Minecraft server
- **reqid** (*int*) – Request ID to use, leave as '-1' to generate one based on system time
- **format\_method** (*int*) – Format method to use. You should specify this using the Client constants
- **timeout** (*int*) – Sets the timeout value for socket operations
- **proto\_num** (*int*) – Protocol number to use, can specify which version we are emulating. Defaults to 0, which is the latest.

**get\_stats**(*format\_method: int = -1, return\_packet: bool = False*) → dict | [PINGPacket](#)

Gets stats from the Minecraft server, and preforms a ping operation.

**Parameters**

- **format\_method** (*int*) – Determines the format method we should use. If '-1', then we use the global value. You should use the Client constants to define this.
- **return\_packet** (*bool*) – Determines if we should return the entire packet. If not, return the payload

**Returns**

Dictionary containing stats, or PINGPacket depending on 'return\_packet'

**Return type**dict, *PINGPacket*

New in version 1.1.0.

The 'format\_method' and 'return\_packet' parameters

New in version 1.2.0.

We now automatically stop this client after the operation

**is\_connected()** → bool

Determines if we are connected.

**Returns**

True if connected, False if otherwise.

**Return type**

bool

**ping()** → float

Pings the Minecraft server and calculates the latency.

New in version 1.3.0.

We no longer request ping stats, we only preform the ping operation.

New in version 1.2.0.

We now automatically stop this client after the operation

**Returns**

Time elapsed(in milliseconds).

**Return type**

float

**raw\_send**(*pingnum: int, packet\_type: int, proto: int = -1, host: str = "", port: int = 0, noread: bool = False*)  
→ *PINGPacket*

Creates a PingPacket and sends it to the Minecraft server

**Parameters**

- **pingnum** (*int*) – Pingnumber of the packet(For ping packets only)
- **packet\_type** (*int*) – Type of packet we are working with
- **proto** (*int*) – Protocol number of the Minecraft server(For handshake only)
- **host** (*str*) – Hostname of the Minecraft server(For handshake only)
- **port** (*int*) – Port of the Minecraft server(For handshake only)
- **noread** (*bool*) – Determining if we are expecting a response

**Returns**

PINGPacket if noread if False, otherwise None

**Return type***PINGPacket*, None**start()**

Starts the connection to the Minecraft server. This is called automatically where appropriate, so you shouldn't have to worry about calling this.

**stop()**

Stops the connection to the Minecraft server. This function should always be called when ceasing network communications, as not doing so could cause problems server-side.

**class** `mctools.mclient.QUERYClient`(*host: str, port: int = 25565, reqid: int = -1, format\_method: int = 1, timeout: int = 60*)

Query client, allows for user to interact with the Minecraft server via the Query protocol.

**Parameters**

- **host** (*str*) – Hostname of the Minecraft server
- **port** (*int*) – Port of the Minecraft server
- **reqid** (*int*) – Request ID to use, leave as ‘-1’ to generate one based on system time
- **format\_method** (*int*) – Format method to use. You should specify this using the Client constants
- **timeout** (*int*) – Sets the timeout value for socket operations

**get\_basic\_stats**(*format\_method: int = -1, return\_packet: bool = False*) → dict | [\*QUERYPacket\*](#)

Gets basic stats from the Query server.

**Parameters**

- **format\_method** (*int*) – Determines the format method we should use. If ‘-1’, then we use the global value. You should use the Client constants to define this
- **return\_packet** (*bool*) – Determines if we should return the entire packet. If not, return the payload

**Returns**

Dictionary of basic stats, or [\*QUERYPacket\*](#), depending on ‘return\_packet’.

**Return type**

dict, [\*QUERYPacket\*](#)

New in version 1.1.0.

The ‘format\_method’ and ‘return\_packet’ parameters

**get\_challenge**() → [\*QUERYPacket\*](#)

Gets the challenge token from the Query server. It is necessary to get a token before every operation, as the Query tokens change every 30 seconds.

**Returns**

[\*QueryPacket\*](#) containing challenge token.

**Return type**

[\*QUERYPacket\*](#)

**get\_full\_stats**(*format\_method: int = -1, return\_packet: bool = False*) → dict | [\*QUERYPacket\*](#)

Gets full stats from the Query server.

**Parameters**

- **format\_method** (*int*) – Determines the format method we should use. If ‘-1’, then we use the global value. You should use the Client constants to define this.
- **return\_packet** (*bool*) – Determines if we should return the entire packet. If not, return the payload

**Returns**

Dictionary of full stats, or QUERYPacket, depending on 'return\_packet'.

**Return type**

dict

New in version 1.1.0.

The 'format\_method' and 'return\_packet' parameters

**is\_connected()** → bool

Determines if we are connected. (UPD doesn't really work that way, so we simply return if we have been started).

**Returns**

True if started, False for otherwise.

**Return type**

bool

**raw\_send**(*packet\_type: int, chall: int, reqtype: int = -1*) → *QUERYPacket*

Creates a packet from the given arguments and sends it. Returns a response.

**Parameters**

- **reqtype** (*int*) – Type of packet to send
- **chall** (*int*) – Challenge Token, ignored if not relevant
- **reqtype** – Request type to utilize

**Returns**

QUERYPacket containing response

**Return type**

*QUERYPacket*

Changed in version 1.3.0.

We now identify request type primarily using the 'packet\_type' parameter. You may provide the request type, but otherwise mctools will auto-determine it for you.

Parameter types and order has been changed.

**start()**

Starts the Query object. This is called automatically where appropriate, so you shouldn't have to worry about calling this.

**stop()**

Stops the connection to the Query server. In this case, it is not strictly necessary to stop the connection, as QueryClient uses the UDP protocol. It is still recommended to stop the instance any way, so you can be explicit in your code as to when you are going to stop communicating over the network.

**class** mctools.mclient.**RCONClient**(*host: str, port: int = 25575, reqid: int = -1, format\_method: int = 1, timeout: int = 60*)

RCON client, allows for user to interact with the Minecraft server via the RCON protocol.

**Parameters**

- **host** (*str*) – Hostname of the Minecraft server (Can be an IP address or domain name, anything your computer can resolve)
- **port** (*int*) – Port of the Minecraft server

- **reqid** (*int*) – Request ID to use, leave as ‘-1’ to generate an ID based on system time
- **format\_method** – Format method to use. You should specify this using the Client constants
- **format\_method** – *int*
- **timeout** (*int*) – Sets the timeout value for socket operations

**authenticate**(*password: str*) → *bool*

Convenience function, does the same thing that ‘login’ does, authenticates you with the RCON server.

**Parameters**

**password** (*str*) – Password to authenticate with

**Returns**

True if successful, False if failure

**Return type**

*bool*

**command**(*com: str, check\_auth: bool = True, format\_method: int = -1, return\_packet: bool = False, frag\_check: bool = True, length\_check: bool = True*) → *RCONPacket* | *str*

Sends a command to the RCON server and gets a response.

---

**Note:** Be sure to authenticate before sending commands to the server! Most servers will simply refuse to talk to you if you do not authenticate.

---

**Parameters**

- **com** (*str*) – Command to send
- **check\_auth** (*bool*) – Value determining if we should check authentication status before sending our command
- **format\_method** (*int*) – Determines the format method we should use. If ‘-1’, then we use the global value You should use the Client constants to define this.
- **return\_packet** (*bool*) – Determines if we should return the entire packet. If not, return the payload
- **frag\_check** (*bool*) – Determines if we should check and handle packet fragmentation

**Warning:** Disabling fragmentation checks could lead to instability! Do so at your own risk!

- **length\_check** (*bool*) – Determines if we should check and handle outgoing packet length

**Warning:** Disabling length checks could lead to instability! Do so at your own risk!

**Returns**

Response text from server

**Return type**

*str*, *RCONPacket*

**Raises**

RCONAuthenticationError: If we are not authenticated to the RCON server, and authentication checking is enabled. We also raise this if the server refuses to serve us, regardless of whether auth checking is enabled. RCONMalformedPacketError: If the packet we received is broken or is not the correct packet. RCONLengthError: If the outgoing packet is larger than 1460 bytes

New in version 1.1.0.

The 'check\_auth', 'format\_method', 'return\_packet', and 'frag\_check' parameters

New in version 1.1.2.

The 'length\_check' parameter

**is\_authenticated()** → bool

Determines if we are authenticated.

**Returns**

True if authenticated, False if not.

**Return type**

bool

**is\_connected()** → bool

Determines if we are connected.

**Returns**

True if connected, False if not.

**Return type**

bool

**login(password: str)** → bool

Authenticates with the RCON server using the given password. If we are already authenticated, then we simply return True.

**Parameters**

**password** (*str*) – RCON Password

**Returns**

True if successful, False if failure

**Return type**

bool

**raw\_send(reqtype: int, payload: str, frag\_check: bool = True, length\_check: bool = True)** → *RCONPacket*

Creates a RCON packet based off the following parameters and sends it. This function is used for all networking operations.

We automatically check if a packet is fragmented(We check it's length). If this is the case, then we send a junk packet, and we keep reading packets until the server acknowledges the junk packet. This operation can take some time depending on network speed, so we offer the option to disable this feature, with the risk that their might be stability issues.

We also check if the packet being sent is too big, and raise an exception of this is the case. This can be disabled by passing False to the 'length\_check' parameter, although this is not recommended.

**Warning:** Use this function at you own risk! You should really call the high level functions, as not doing so could mess up the state/connection of the client.



**Parameters**

- **reqtype** (*int*) – Request type
- **payload** (*str*) – Payload to send
- **frag\_check** (*bool*) – Determines if we should check and handle packet fragmentation.

**Warning:** Disabling fragmentation checks could lead to instability! Do so at your own risk!

**Parameters**

- **length\_check** (*bool*) – Determines if we should check for outgoing packet length

**Warning:** Disabling length checks could lead to instability! Do so at your own risk!

**Returns**

RCONPacket containing response from server

**Return type**

*RCONPacket*

**Raises**

RCONAuthenticationError: If the server refuses to server us and we are not authenticated.  
 RCONMalformedPacketError: If the request ID's do not match of the packet is otherwise malformed.

New in version 1.1.0.

The 'frag\_check' parameter

New in version 1.1.2.

The 'length\_check' parameter

**start()**

Starts the connection to the RCON server. This is called automatically where appropriate, so you shouldn't have to worry about calling this.

**stop()**

Stops the connection to the RCON server. This function should always be called when ceasing network communications, as not doing so could cause problems server-side.

## 8.3 Packet Reference

Packet classes to hold RCON and Query data

**class** mctools.packet.**BasePacket**

Parent class for packet implementations.

**classmethod** **from\_bytes**(*byts: bytes*) → Any

Classmethod to create a packet from bytes. Rises an NotImplementedError exception, as this function should be overridden in the child class.

**Parameters**

**byts** (*bytes*) – Bytes from remote entity

**Returns**

Packet object

```
class mctools.packet.PINGPacket(pingnum: int, packet_type: int, data: dict | None = None, proto: int = 0,
                                host: str = "", port: int = 0)
```

Class representing a Server List Ping Packet(Or ping for short).

PINGPackets have the following types:

- HANDSHAKE - Initial handshake with server, this is always the first operation!
- STATUS\_REQUEST - Client is asking server for status information
- STATUS\_RESPONSE - Server responds to status request with status response, which contains server info
- PING\_REQUEST - Client is asking server to respond with a pong packet, useful for determining latency
- PONG\_RESPONSE - Server responds to ping request with pong response

The lifecycle of a ping connection is as follows:

1. Handshake is initiated by client, which MUST be preformed before any operations!
2. Client may request server status information, server responds. Optionally, clients may skip this step
3. Client makes a ping request, server responds with pong response, connection is closed

**Parameters**

- **pingnum** (*int*) – Number to use for ping operations
- **packet\_type** (*int*) – Type of packet we are working with
- **data** (*dict*, *None*) – Ping data from Minecraft server
- **proto** (*str*, *None*) – Protocol Version used
- **host** (*str*, *None*) – Hostname of the Minecraft server
- **port** (*int*) – Port number of the Minecraft server

```
classmethod from_bytes(byts: bytes) → PINGPacket
```

Creates a PINGPacket from bytes.

**Parameters**

**byts** (*bytes*) – Bytes to decode

**Returns**

PINGPacket decoded from bytes

**Return type**

*PINGPacket*

```
class mctools.packet.QUERYPacket(packet_type: int, reqid: int, chall: int, data: dict | None = None, reqtype:
                                int = -1)
```

Class representing a Query Packet.

QUERYPackets have the following types:

- BASIC\_REQUEST - Client is requesting basic data
- BASIC\_RESPONSE - Server is responding with basic data

- `FULL_REQUEST` - Client is requesting full data
- `FULL_RESPONSE` - Server is responding with basic data
- `HANDSHAKE_REQUEST` - Client is requesting to handshake
- `HANDSHAKE_RESPONSE` - Server has acknowledged the handshake

The lifecycle of a ping connection is as follows:

1. Client requests to handshake, and the server acknowledges and provides a challenge token
2. **Once handshake has completed, clients may preform one of the two actions using provided challenge token:**
  - a. Client may request basic server stats
  - b. Client may request full server stats
3. Finally, server responds with stats at the requested level, be it basic or full

This packet has some optional parameters, namely 'reqtype'. This parameter, if unspecified (has a value of -1), will be auto-determined at runtime. Users may set this parameter, and decoded packets will have this parameter set, but it is recommended to utilize 'packet\_type' to identify packets instead.

#### Parameters

- **packet\_type** (*int*) – Type of this Query packet
- **reqid** (*int*) – Request ID of the Query packet
- **chall** (*int*) – Challenge token from Query server, -1 if N/A
- **data** (*dict*) – Data from the Query server
- **reqtype** (*int*) – Type of request, this will be auto-determined at runtime

Changed in version 1.3.0.

Removed the 'data\_type' attribute, and internally it is no longer used. We provide the compatibility property of the same name that behaves the same as the 'data\_type' attribute.

Packets now use 'packet\_type' to identify themselves instead of 'reqtype'

#### property data\_type: str

Determines the data type using the request type.

This property is here to provide compatibility with components that may utilize this value, as it was removed.

#### Returns

'basic' for basic data, 'full' for full data, blank string for neither

#### Return type

str

#### classmethod from\_bytes(*byts: bytes*) → *QUERYPacket*

Creates a Query packet from bytes.

#### Parameters

**byts** (*bytes*) – Bytes from Query server

#### Returns

Created *QUERYPacket*

#### Return type

*QUERYPacket*

**is\_basic()** → bool

Determines if this packet contains (or is asking for) basic server information.

**Returns**

True if basic, False if not

**Return type**

bool

New in version 1.3.0.

Convenience method for identifying basic packets

**is\_full()** → bool

Determines if this packet contains (or is asking for) full server information.

**Returns**

True if full, False if not

**Return type**

bool

New in version 1.3.0.

Convenience method for identifying full packets

**class** `mctools.packet.RCONPacket`(*reqid: int, reqtype: int, payload: str, length: int = 0*)

Class representing a RCON packet.

**Parameters**

- **reqid** (*int*) – Request ID of the RCON packet
- **reqtype** (*int*) – Request type of the RCON packet
- **payload** (*str*) – Payload of the RCON packet

Changed in version 1.3.0.

Packet types are now defined here, instead of RCONProtocol

**classmethod** **from\_bytes**(*byts: bytes*) → *RCONPacket*

Creates a RCON packet from bytes.

**Parameters**

**byts** (*bytes*) – Bytes from RCON server

**Returns**

RCONPacket created from bytes

**Return type**

*RCONPacket*

## 8.4 Encoding Reference

Encoding/Decoding components for RCON and Query.

**class** `mctools.encoding.PINGEncoder`

Ping encoder/decoder.

**static** `decode(bytes: bytes) → Tuple[dict, int]`

Decodes a ping packet. NOTE: Bytes provided should NOT include the length, that is interpreted and used by the PINGProtocol.

**Parameters**

**bytes** (*bytes*) – Bytes to decode

**Returns**

Tuple of decoded values, data and packet type

**Return type**

Tuple[dict, str]

**static** `decode_sock(sock) → int`

Decodes a var(int/long) of variable length or value. We use a socket to continuously pull values until we reach a valid value, as the length of these var(int/long)s can be either very long or very short.

**Parameters**

**sock** (*socket.socket*) – Socket object to get bytes from.

**Returns**

Decoded integer

**Return type**

int

**static** `decode_varint(bytes: bytes) → Tuple[int, int]`

Decodes varint bytes into an integer. We also return the amount of bytes read.

**Parameters**

**bytes** (*bytes*) – Bytes to decode

**Returns**

result, bytes read

**Return type**

int, int

**static** `encode(data: dict, pingnum: int, packet_type: int, proto: int, hostname: str, port: int) → bytes`

Encodes a Ping packet.

**Parameters**

- **data** (*dict*) – Ping data from server
- **pingnum** (*int*) – Number for ping operations
- **packet\_type** (*int*) – Type of packet we are working with
- **proto** (*int*) – Protocol version number
- **hostname** – Hostname of the minecraft server

:type hostname :param port: Port we are connecting to :type port: int :return: Encoded data :rtype: bytes

Changed in version 1.3.0.

We now raise an exception if invalid packet type is provided

**static** **encode\_varint**(*num: int*) → bytes

Encodes a number into varint bytes.

**Parameters**

**num** (*int*) – Number to encode

**Returns**

Encoded bytes

**Return type**

bytes

**class** **mctools.encoding.QUERYEncoder**

Query encoder/decoder

**static** **decode**(*byts: bytes*) → Tuple[int, int, int, dict, int]

Decodes packets from the Query protocol. This gets really messy, as the payload can be many different things. We also figure out what kind of data we are working with, and return that.

**Parameters**

**byts** (*bytes*) – Bytes to decode

**Returns**

Tuple containing decoded items - packet\_type, reqid, chall, data, reqtype

**Return type**

Tuple[int, int, int, dict, str]

Changed in version 1.3.0.

Changed the return values

**static** **encode**(*packet\_type: int, reqtype: int, reqid: int, chall: int*) → bytes

Encodes a Query Packet.

**Parameters**

- **packet\_type** (*int*) – Type of query packet
- **reqtype** (*int*) – Type of request, will be auto-determined if (-1)
- **reqid** (*int*) – Request ID of query packet
- **chall** (*int*) – Challenge token from query server

**Returns**

Encoded data

**Return type**

bytes

We no longer consider the 'data\_type', we only do checking using the packet type.

If reqtype is not provided (-1), we determine the value using the packet\_type parameter.

Input parameters have been changed

**class** **mctools.encoding.RCONEncoder**

RCON Encoder/Decoder.

**static decode**(*byts: bytes*) → Tuple[int, int, str]

Decodes raw bytestring packet from the RCON server. NOTE: DOES NOT DECODE LENGTH! The 'length' value is omitted, as it is interpreted by the RCONProtocol. If your bytestring contains the encoded length, simply remove the first 4 bytes.

**Parameters**

**byts** (*bytes*) – Bytestring to decode

**Returns**

Tuple containing values, request ID, request type, payload

**Return type**

Tuple[int, int, str]

**static encode**(*reqid: int, reqtype: int, payload: str*) → bytes

Encodes a RCON packet.

**Parameters**

- **reqid** (*int*) – Request ID of RCON packet
- **reqtype** (*int*) – Request type of RCON packet
- **payload** (*str*) – Payload of RCON packet

**Returns**

Bytestring of encoded data

**Return type**

bytes

## 8.5 Protocol Reference

Low-level protocol stuff for RCON, Query and Server List Ping.

**class** `mctools.protocol.BaseProtocol`

Parent Class for protocol implementations. Every protocol instance should inherit this class!

**read**() → Any

Method to receive data from remote entity This raises a `NotImplementedError`, as it should be overridden in the child class.

**Returns**

Data received, data type is arbitrary

**read\_tcp**(*length: int*) → bytes

Reads data over the TCP protocol.

**Parameters**

**length** (*int*) – Amount of data to read

**Returns**

Read bytes

**Return type**

bytes

**read\_udp()** → Tuple[bytes, str]

Reads data over the UDP protocol.

**Returns**

Bytes read and address

**Return type**

Tuple[bytes, str]

**send(pack: Any)**

Method to send data to remote entity, data type is arbitrary. This raises a `NotImplementedErrorException`, as it should be overridden in the child class.

**Parameters**

**data** – Some data in some format.

**set\_timeout(timeout: int)**

Sets the timeout value for the underlying socket object.

**Parameters**

**timeout (int)** – Value in seconds to set the timeout to

New in version 1.1.0.

This method now works before the protocol object is started

**start()**

Method to start the connection to remote entity.

We simply set the connected value, and configure the timeout.

**stop()**

Method to stop the connection to remote entity.

We simply set the connected value.

**write\_tcp(byts: bytes)**

Writes data over the TCP protocol.

**Parameters**

**byts (bytes)** – Bytes to send

**write\_udp(byts: bytes, host: str, port: int)**

Writes data over the UPD protocol.

**Parameters**

- **byts (bytes)** – Bytes to write
- **host (str)** – Hostname of remote host
- **port (int)** – Portname of remote host

**class mctools.protocol.PINGProtocol(host: str, port: int, timeout: int)**

Protocol implementation for server list ping - uses TCP sockets.

**Parameters**

- **host (str)** – Hostname of the Minecraft server
- **port (int)** – Port number of the Minecraft server
- **timeout (int)** – Timeout value used for socket operations



**read()** → *PINGPacket*

Read data from the Minecraft server and convert it into a PINGPacket.

**Returns**

PINGPacket

**Return type**

*PINGPacket*

**send(pack: PINGPacket)**

Sends a ping packet to the Minecraft server.

**Parameters**

**pack** (*PINGPacket*) – PINGPacket

**start()**

Starts the connection to the Minecraft server.

**stop()**

Stopping the connection to the Minecraft server.

**class** `mctools.protocol.QUERYProtocol`(*host: str, port: int, timeout: int*)

Protocol implementation for Query - Uses UDP sockets.

**Parameters**

- **host** (*str*) – The hostname of the Query server.
- **port** (*int*) – Port number of the Query server
- **timeout** (*int*) – Timeout value for socket operations

**read()** → *QUERYPacket*

Gets a Query packet from the Query server.

**Returns**

QUERYPacket

**Return type**

*QUERYPacket*

**send(pack: QUERYPacket)**

Sends a packet to the Query server.

**Parameters**

**pack** (*QUERYPacket*) – Packet to send

**start()**

Sets the protocol object as ready to communicate.

**stop()**

Sets the protocol object as not ready to communicate.

**class** `mctools.protocol.RCONProtocol`(*host: str, port: int, timeout: int*)

Protocol implementation for RCON - Uses TCP sockets.

**Parameters**

- **host** (*str*) – Hostname of RCON server
- **port** (*int*) – Port number of the RCON server
- **timeout** (*int*) – Timeout value for socket operations.

Changed in version 1.3.0.

Moved packet types to RCONPacket

**read()** → *RCONPacket*

Gets a RCON packet from the RCON server.

**Returns**

RCONPacket containing payload

**Return type**

*RCONPacket*

**send(pack: RCONPacket, length\_check: bool = False)**

Sends a packet to the RCON server.

**Parameters**

- **pack** (*RCONPacket*) – RCON Packet
- **length\_check** (*bool*) – Boolean determining if we should check packet length

New in version 1.1.2.

The 'length\_check' parameter

**start()**

Starts the connection to the RCON server.

**stop()**

Stops the connection to the RCON server.

## 8.6 Formatting Tools Reference

Formatters to alter response output - makes everything look pretty

**class** `mctools.formattertools.BaseFormatter`

Parent class for formatter implementations.

**static clean**(*text: Any*) → *Any*

Removes format chars, instead of replacing them with actual values. Great for if the user does not want/have color support, And wants to remove the unneeded format chars.

**Parameters**

**text** (*str*) – Text to be formatted

**Returns**

Formatted text

**Return type**

*str*

**static format**(*text: Any*) → *Any*

Formats text in any way fit. Note: This should not remove format chars, that's what remove is for, Simply replace them with their required values.

**Parameters**

**text** (*str*) – Text to be formatted

**Returns**

Formatted text

**Return type**

str

**static** `get_id()` → int

Should return an integer representing the formatter ID. This is important, as it determines how the formatters are sorted. Sorting goes from least - greatest, meaning that formatters with a lower ID get executed first.

**Returns**

Formatter ID

**Return type**

int

**class** `mctools.formattertools.ChatObjectFormatter`

A formatter that handles the formatting scheme of ChatObjects. This description scheme differs from primitive chat objects, as it doesn't use formatting characters. It instead uses a collection of dictionaries to define what colors and attributes should be used.

**static** `clean(text: dict)` → str

Cleans a description directory, and returns the cleaned text.

**Parameters****text** (*dict*) – Dictionary to be formatted**Returns**

Cleaned text

**Return type**

str

**static** `format(chat: dict, color: str = "", attrib: str = "")` → str

Formats a description dictionary, and returns the formatted text. This gets tricky, as the server could define a variable number of child dicts that have their own attributes/extra content. So, we must implement some recursion to be able to parse and understand the entire string.

**Parameters**

- **chat** (*dict*) – Dictionary to be formatted
- **color** (*str*) – Parent text color, only used in recursive operations
- **attrib** (*str*) – Parent attributes, only used in recursive operations

**Returns**

Formatted text

**Return type**

str

**class** `mctools.formattertools.DefaultFormatter`**Formatter with good default operation:**

- `format()` - Replaces all formatter codes with ascii values
- `clean()` - Removes all formatter codes

This formatter ONLY handles formatting codes and text attributes.

**static** `clean(text: str)` → str

Removes all format codes. Does not use color/text effects.

**Parameters****text** (*str*) – Text to be formatted.

**Returns**

Formatted text.

**Return type**

str

**static** **format**(*text: str*) → str

Replaces all format codes with their intended values (Color, text effect, ect).

**Parameters**

**text** (*str*) – Text to be formatted.

**Returns**

Formatted text.

**Return type**

str

**static** **get\_id**() → int

Returns this formatters ID, which is 10.

**Returns**

Formatter ID

**Return type**

int

**class** mctools.formattertools.**FormatterCollection**

A collection of formatters - Allows for formatting text with multiple formatters, and determining which formatter is relevant to the text.

This class offers the following constants:

- `FormatterCollection.QUERY` - Used for identifying Query protocol content.
- `FormatterCollection.PING` - Used for identifying Server List Ping protocol content.

**add**(*form: BaseFormatter*, *command: str | Iterable[str] = ""*, *ignore: str | Iterable[str] = ""*) → bool

Adds a formatter, MUST inherit the BaseFormatter class. Your formatter must either be instantiated upon adding it, or the 'clean' and 'format' methods are static, as FormatterCollection will not attempt to instantiate your object.

**Parameters**

- **form** (*BaseFormatter*) – Formatter to add.
- **command** (*Union[str, Iterable[str]]*) – Command(s) to register the formatter with. May be a string or iterable. Supply an empty string ('') to affiliate with every command, or an iterable to affiliate with multiple.
- **ignore** (*Union[str, Iterable[str]]*) – Commands to ignore, formatter will not format them, leave blank to accept everything. May be a string or iterable. Supply an empty string ('') to allow all commands, or an iterable to affiliate with multiple.

**Returns**

True for success, False for Failure.

**Return type**

bool

Changed in version 1.3.0.

We not accept blanks strings instead of None to denote a global match/global allow

**clean**(*text: Any, command: str*) → Any

Runs the text through the clean() method of every formatter. These formatters will remove characters, without replacing them, Most likely format chars. We only send non-string data to specific formatters.

**Parameters**

- **text** (*str*) – Text to be formatted.
- **command** (*str*) – Command issued - determines which formatters are relevant. You may leave command blank to affiliate with every formatter.

**Returns**

Formatted text.

**Return type**

str

**clear**()

Removes all formatters from the list.

**format**(*text: Any, command: str*) → Any

Runs the text through the format() function of relevant formatters. These formatters will be removing and replacing characters, Most likely format chars.

**Parameters**

- **text** (*str*) – Text to be formatted
- **command** (*str*) – Command issued - determines which formatters are relevant. You may leave command blank to affiliate with every formatter.

**Returns**

Formatted text.

**Return type**

str

**get**() → List[Tuple[BaseFormatter, str | Iterable[str], str | Iterable[str]]]

Returns the list of formatters. Can be used to check loaded formatters, or manually edit list. Be aware, that manually editing the list means that the formatters may have some unstable operations.

**Returns**

List of formatters and commands

**Return type**

List[Tuple[BaseFormatter, Command, Command]]

**remove**(*form: BaseFormatter*) → bool

Removes a specified formatter from the list.

**Parameters**

**form** (BaseFormatter) – Formatter to remove

**Returns**

True on success, False on failure

**Return type**

bool

**class** mctools.formattertools.PINGFormatter

Formatter for formatting responses from the server via Server List Ping protocol. We only format relevant content, such as description and player names. We also use special formatters, such as ChatObjectFormatter, and SampleDescriptionFormatter.

**static clean**(*stat\_dict: dict*) → dict

Cleaned a dictionary of stats from the Minecraft server.

**Parameters**

**stat\_dict** (*dict*) – Dictionary to format

**Returns**

Formatted statistics dictionary.

**Return type**

dict

**static format**(*stat\_dict: dict*) → dict

Formats a dictionary of stats from the Minecraft server.

**Parameters**

**stat\_dict** (*dict*) – Dictionary to format

**Returns**

Formatted statistics dictionary.

**Return type**

dict

**class** mctools.formattertools.**QUERYFormatter**

Formatter for formatting responses from the Query server. Will only format certain parts, as servers SHOULD follow a specific implementation for Query.

**static clean**(*text: dict*) → dict

Removes format chars from the response.

**Parameters**

**text** (*dict*) – Response from Query server

**Returns**

Formatted content.

**Return type**

dict

**static format**(*text: dict*) → dict

Replaces format chars with actual values.

**Parameters**

**text** (*dict*) – Response from Query server(Ideally in dict form).

**Returns**

Formatted content.

**Return type**

dict

**class** mctools.formattertools.**SampleDescriptionFormatter**

Some servers like to put special messages in the ‘sample players’ field. This formatter attempts to handle this, and sort valid players and null players into separate categories.

**static clean**(*text: List[dict]*) → Tuple[List[str], str]

Does the same operation as format. This is okay because we don’t change up any values or alter the content, we just organize them, and the color formatter will handle it later.

**Parameters**

**text** (*List[dict]*) – Valid players, Message encoded in the sample list

**Returns**

List containing valid users, and message.

**static format**(*text: List[dict]*) → Tuple[List[str], str]

Formats a sample list of users, removing invalid ones and adding them to a message sublist. We return the message in the playerlist, and also return valid players.

**Parameters**

**text** (*List[dict]*) – List of valid players, Message encoded in the sample list

**Returns**

List containing valid users, any embedded messages

**Return type**

Tuple[List[str], str]





## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### m

- `mctools.encoding`, 49
- `mctools.errors`, 37
- `mctools.formattertools`, 54
- `mctools.mclient`, 38
- `mctools.packet`, 45
- `mctools.protocol`, 51



## A

`add()` (*mctools.formattertools.FormatterCollection* method), 56  
`authenticate()` (*mctools.mclient.RCONClient* method), 43

## B

`BaseClient` (class in *mctools.mclient*), 38  
`BaseFormatter` (class in *mctools.formattertools*), 54  
`BasePacket` (class in *mctools.packet*), 45  
`BaseProtocol` (class in *mctools.protocol*), 51

## C

`ChatObjectFormatter` (class in *mctools.formattertools*), 55  
`clean()` (*mctools.formattertools.BaseFormatter* static method), 54  
`clean()` (*mctools.formattertools.ChatObjectFormatter* static method), 55  
`clean()` (*mctools.formattertools.DefaultFormatter* static method), 55  
`clean()` (*mctools.formattertools.FormatterCollection* method), 56  
`clean()` (*mctools.formattertools.PINGFormatter* static method), 57  
`clean()` (*mctools.formattertools.QUERYFormatter* static method), 58  
`clean()` (*mctools.formattertools.SampleDescriptionFormatter* static method), 58  
`clear()` (*mctools.formattertools.FormatterCollection* method), 57  
`Client` (built-in class), 7  
`command()` (*mctools.mclient.RCONClient* method), 43

## D

`data_type` (*mctools.packet.QUERYPacket* property), 47  
`decode()` (*mctools.encoding.PINGEncoder* static method), 49  
`decode()` (*mctools.encoding.QUERYEncoder* static method), 50  
`decode()` (*mctools.encoding.RCONEncoder* static method), 50

`decode_sock()` (*mctools.encoding.PINGEncoder* static method), 49  
`decode_varint()` (*mctools.encoding.PINGEncoder* static method), 49  
`DefaultFormatter` (class in *mctools.formattertools*), 55

## E

`encode()` (*mctools.encoding.PINGEncoder* static method), 49  
`encode()` (*mctools.encoding.QUERYEncoder* static method), 50  
`encode()` (*mctools.encoding.RCONEncoder* static method), 51  
`encode_varint()` (*mctools.encoding.PINGEncoder* static method), 50

## F

`format()` (*mctools.formattertools.BaseFormatter* static method), 54  
`format()` (*mctools.formattertools.ChatObjectFormatter* static method), 55  
`format()` (*mctools.formattertools.DefaultFormatter* static method), 56  
`format()` (*mctools.formattertools.FormatterCollection* method), 57  
`format()` (*mctools.formattertools.PINGFormatter* static method), 58  
`format()` (*mctools.formattertools.QUERYFormatter* static method), 58  
`format()` (*mctools.formattertools.SampleDescriptionFormatter* static method), 59  
`FormatterCollection` (class in *mctools.formattertools*), 56  
`from_bytes()` (*mctools.packet.BasePacket* class method), 45  
`from_bytes()` (*mctools.packet.PINGPacket* class method), 46  
`from_bytes()` (*mctools.packet.QUERYPacket* class method), 47  
`from_bytes()` (*mctools.packet.RCONPacket* class method), 48

## G

gen\_reqid() (*mctools.mclient.BaseClient* method), 38  
 get() (*mctools.formattertools.FormatterCollection* method), 57  
 get\_basic\_stats() (*mctools.mclient.QUERYClient* method), 41  
 get\_challenge() (*mctools.mclient.QUERYClient* method), 41  
 get\_formatter() (*mctools.mclient.BaseClient* method), 38  
 get\_full\_stats() (*mctools.mclient.QUERYClient* method), 41  
 get\_id() (*mctools.formattertools.BaseFormatter* static method), 55  
 get\_id() (*mctools.formattertools.DefaultFormatter* static method), 56  
 get\_stats() (*mctools.mclient.PINGClient* method), 39

## I

is\_authenticated() (*mctools.mclient.RCONClient* method), 44  
 is\_basic() (*mctools.packet.QUERYPacket* method), 47  
 is\_connected() (*mctools.mclient.BaseClient* method), 38  
 is\_connected() (*mctools.mclient.PINGClient* method), 40  
 is\_connected() (*mctools.mclient.QUERYClient* method), 42  
 is\_connected() (*mctools.mclient.RCONClient* method), 44  
 is\_full() (*mctools.packet.QUERYPacket* method), 48

## L

login() (*mctools.mclient.RCONClient* method), 44

## M

mctools.encoding  
     module, 49  
 mctools.errors  
     module, 37  
 mctools.formattertools  
     module, 54  
 mctools.mclient  
     module, 38  
 mctools.packet  
     module, 45  
 mctools.protocol  
     module, 51  
 MCToolsError, 37  
 module  
     mctools.encoding, 49  
     mctools.errors, 37  
     mctools.formattertools, 54

mctools.mclient, 38  
 mctools.packet, 45  
 mctools.protocol, 51

## P

ping() (*mctools.mclient.PINGClient* method), 40  
 PINGClient (class in *mctools.mclient*), 39  
 PINGEncoder (class in *mctools.encoding*), 49  
 PINGError, 37  
 PINGFormatter (class in *mctools.formattertools*), 57  
 PINGMalformedPacketError, 37  
 PINGPacket (class in *mctools.packet*), 46  
 PINGProtocol (class in *mctools.protocol*), 52  
 ProtocolError, 37  
 ProtoConnectionClosed, 37

## Q

QUERYClient (class in *mctools.mclient*), 41  
 QUERYEncoder (class in *mctools.encoding*), 50  
 QUERYFormatter (class in *mctools.formattertools*), 58  
 QUERYPacket (class in *mctools.packet*), 46  
 QUERYProtocol (class in *mctools.protocol*), 53

## R

raw\_send() (*mctools.mclient.PINGClient* method), 40  
 raw\_send() (*mctools.mclient.QUERYClient* method), 42  
 raw\_send() (*mctools.mclient.RCONClient* method), 44  
 RCONAuthenticationError, 37  
 RCONClient (class in *mctools.mclient*), 42  
 RCONCommunicationError, 37  
 RCONEncoder (class in *mctools.encoding*), 50  
 RCONError, 38  
 RCONLengthError, 38  
 RCONMalformedPacketError, 38  
 RCONPacket (class in *mctools.packet*), 48  
 RCONProtocol (class in *mctools.protocol*), 53  
 read() (*mctools.protocol.BaseProtocol* method), 51  
 read() (*mctools.protocol.PINGProtocol* method), 52  
 read() (*mctools.protocol.QUERYProtocol* method), 53  
 read() (*mctools.protocol.RCONProtocol* method), 54  
 read\_tcp() (*mctools.protocol.BaseProtocol* method), 51  
 read\_udp() (*mctools.protocol.BaseProtocol* method), 51  
 remove() (*mctools.formattertools.FormatterCollection* method), 57

## S

SampleDescriptionFormatter (class in *mctools.formattertools*), 58  
 send() (*mctools.protocol.BaseProtocol* method), 52  
 send() (*mctools.protocol.PINGProtocol* method), 53  
 send() (*mctools.protocol.QUERYProtocol* method), 53

`send()` (*mctools.protocol.RCONProtocol method*), 54  
`set_timeout()` (*mctools.mclient.BaseClient method*), 39  
`set_timeout()` (*mctools.protocol.BaseProtocol method*), 52  
`start()` (*mctools.mclient.BaseClient method*), 39  
`start()` (*mctools.mclient.PINGClient method*), 40  
`start()` (*mctools.mclient.QUERYClient method*), 42  
`start()` (*mctools.mclient.RCONClient method*), 45  
`start()` (*mctools.protocol.BaseProtocol method*), 52  
`start()` (*mctools.protocol.PINGProtocol method*), 53  
`start()` (*mctools.protocol.QUERYProtocol method*), 53  
`start()` (*mctools.protocol.RCONProtocol method*), 54  
`stop()` (*mctools.mclient.BaseClient method*), 39  
`stop()` (*mctools.mclient.PINGClient method*), 40  
`stop()` (*mctools.mclient.QUERYClient method*), 42  
`stop()` (*mctools.mclient.RCONClient method*), 45  
`stop()` (*mctools.protocol.BaseProtocol method*), 52  
`stop()` (*mctools.protocol.PINGProtocol method*), 53  
`stop()` (*mctools.protocol.QUERYProtocol method*), 53  
`stop()` (*mctools.protocol.RCONProtocol method*), 54

## W

`write_tcp()` (*mctools.protocol.BaseProtocol method*), 52  
`write_udp()` (*mctools.protocol.BaseProtocol method*), 52