# hisock

*Release 2.0.1*

**SSS-Says-Snek**

**Oct 24, 2022**

# CONTENTS:

`hisock` (also called `Hisock` and `HiSock`) is a higher-level extension of the *socket* module, with simpler and more efficient usages. It completely removes the hassle of headers, and utilizes decorators to maintain a good code structure.

---

**Note:** This project is still currently under development. This is also the first package of the developer, SSS-Says-Snek, so if you find something that needs improvement, submit an issue on the GitHub Repository

---

CHAPTER

# ONE

# TABLE OF CONTENTS

- genindex
- modindex
- search

## 1.1 Quickstart

These pages provide you with information to start your HiSock journey!

Follow the *Beginner Tutorial* for a basic tutorial on how to use HiSock.

### 1.1.1 Understanding `HiSock`

**Table of Contents**



#### What is `HiSock`? What is it not?

The TL;DR of this all is:

`HiSock` is a higher-level extension of the socket Python module with simpler and more efficient uses.

`HiSock` is not a replacement for sockets nor a new network protocol. It doesn't work through a "request/response" method.

Now, if you're new to this, most likely you will have no idea what any of this means. But, that's okay! The goal of this section is to help you understand.

#### What is the `socket` module? How does it relate to `HiSock`?

Let's focus on what `HiSock` is built on, `socket`.

`socket` is a low-level networking interface that uses "sockets" to communicate between computers over a network. In sockets, when you send data, you need to send a "header" first. Before you even send your data, you need to send some other data telling how long that data is. Along with that, `socket` is pretty bare bone. So, I developed `HiSock`, which simplifies `socket` down and provides additional features.

`HiSock` focuses on abstracting the complex parts of sockets so you can focus on what you actually want to do. It does this through managing headers on its own, "type-casting" everything so you don't have to convert data to bytes and back, event-driven architecture with decorators, threading, data streams, and more. Basically, it's everything `socket` can do, minus the boilerplate code you have to write every time.

#### How do you send and receive data with `HiSock`?

As `HiSock` is an event-driven module, data sent must have a *command* (also known as *event*) before the data (not required). An example of this could be if the command is `send_number` and the data is `847`.

But what data can be sent? Sockets work with bytes, but most times, you don't just want bytes. As touched on earlier, `HiSock` comes with a type-cast system, which will convert stuff to bytes under-the-hood, so you can focus on sending the data you want and not having to deal with converting it yourself.

#### Conclusion

Now that you understand *what* `HiSock` is and why it exists a little better, you can finally start to the *Tutorials*!

### 1.1.2 Installation

To install `HiSock`, you will need to make sure you have PIP and Python3.7 or later installed and added to your PATH (ensure `pip` and `python` works). There are two main ways to install `HiSock`. You can install it through PyPi (recommended) or you can install it from the GitHub.

> **Warning:** `HiSock` is the first project I ever published to PyPI, so there might be some quirks on PyPI here and there, like the sudden burst of version post-releases. However, I will try to keep this at the bare minimum, and hopefully figure out PyPI good enough.

#### Installing via PyPI

The recommended way to install a stable version of `HiSock` is through PyPI. To install hisock, open a terminal/command prompt and type:

```
$ python -m pip install hisock (Windows)
    OR
$ pip install hisock (Windows)
    OR
$ python3 -m pip install hisock (Mac/Linux)
    OR
$ pip3 install hisock (Mac/Linux)
```

You've now successfully installed a stable version of `HiSock`!

#### Installing via GitHub

Sometimes, however, you might want to install the *latest* version of `HiSock`, not just the *stable* version. To do this, you can either download the repository from GitHub here or you can clone the repository (recommended) via Git.

To install from Git, open a terminal/command prompt and type:

```
$ git clone https://github.com/SSS-Says-Snek/hisock.git (Git)
    OR
$ gh repo clone SSS-Says-Snek/hisock (GitHub CLI)
```

Or, you can go onto GitHub and download the ZIP file by pressing the green "Code" button, then clicking "Download ZIP". You can then unzip the file into a folder with your favorite unzip tool.

Then, go back to the terminal or command prompt and type:

```
$ cd hisock
```

After you're now in the working directory of the repo, install it in editable mode with:

```
$ pip install -e .
```

You should now have successfully installed the latest version of `HiSock`! If this doesn't work, then try one of the alternatives in Method 1, but replace `hisock` with `-e .` (E.g `python3 -m pip install -e .`)

---

**Note:** If you want to check if `HiSock` is *actually* installed, run this command in your terminal or command prompt:

```
$ python -c $'try:\n\timport hisock;print(f"Hisock {hisock.constants.__version__}␣
↪successfully installed")\nexcept Exception as e:print(f"Failed to install hisock for
↪{e} reason")'
```

---

## 1.2 Tutorials

### 1.2.1 Beginner Tutorial

**Table of Contents**

### Before you start

It is highly encouraged that you read the *Understanding HiSock* section before reading the tutorial.

This tutorial also relies on you having basic networking knowledge (client/server, IP addresses, ports, etc.) and at least basic Python knowledge.

Lastly, if you don't have `HiSock` installed, do that now! Read *Installation*.

This tutorial will focus on:

- Creating a server
- Creating a client
- Sending and receiving data
- Acting upon the data

---

**Note:** For this tutorial, I will be referring the IP addresses as `hisock.utils.get_local_ip()`. In reality, you will most likely use a hard-coded IP address from a user input.

---

---

**Note:** When I refer to "a way to identify the client", I am talking about either:

- a tuple of the IP address and port of the client or
- a string of the client's name (this will have ambiguity if multiple clients share the same name)

---

---

**Note:** There are a few terms that are used interchangeably here.

- Message, content, and data mean the same thing.
- Command and event mean the same thing.

---

Now, without further ado, let's begin!

---

### Creating our first server

In `HiSock`, there exists a class, `HiSockServer`, in the `server` module. To create a server, you will need to create an instance of this class. The `__init__` function of the `HiSockServer` class takes a required tuple parameter, which is the IP address as a string and port as an integer to start the server on. To find the local IP address, there is a function called `utils.get_local_ip()`. For the port, a number between 1024 and 65535 *should* be fine.

`HiSockServer` instances have a `start()` method to them, which will allow the server to listen for commands and data being sent.

```
import hisock

server = hisock.server.HiSockServer((hisock.utils.get_local_ip(), 6969))

server.start()
```

That's basically it! Of course, this server is useless, but hey, it's a step in the right direction! We'll add on to this later on.

Obviously, without a client, a server is kind of pointless. So, let's spice things up with some client code!

---

### Creating our first client

In `HiSock`, there is a class, `HiSockClient`, in the `client` module. To create a client, you will need to create an instance of this class. The `__init__` function of the `HiSockClient` class takes two required parameters. The first parameter is a tuple with the IP address and port that the server is running on. The second (optional) parameter is for the name of the client, which is used for identification. The third (optional) parameter is the group of the client, which won't be talked about in this tutorial.

Like `HiSockServer`, `HiSockClient` needs to have its `start()` method called to start the client.

```
import hisock

client = hisock.client.HiSockClient(
    (hisock.utils.get_local_ip(), 6969),
    name=input("What is your name? >")
)

client.start()
```

Like the server, this doesn't do anything at all yet. Next, we will explore sending and receiving data in an example.

---

### Transmitting Data

Let's explore transmitting data for `HiSock`!

`HiSock` is an event-driven module, and as such, has an `on` decorator and `send()` methods for both `HiSockClient` and `HiSockServer`.

---

### Receiving data

---

**Note:** There is another way of receiving data, which is the `recv()` method. This is not covered in this tutorial, but it is covered in the *Intermediate-level Tutorial*.

---

When a function is prefaced with the `on` decorator, it will run on something. It will listen for a command and run when that command is received.

The on decorator takes a maximum of three parameters. One of the parameters is the command to listen on. The second (optional) parameter is whether to run the listener in its own thread or not. The third (optional) parameter is whether to override a reserved command, and this tutorial won't be covering it.

For the server: The on decorator will send a maximum of two parameters to the function it is decorating (there are a few exceptions we will touch on). The first parameter is the client data. It is an instance of `ClientInfo` that includes the client's name, client IP address, and the group the client is in (can be type-casted to a dict). The second parameter is the data that is being received.

For the client: the on decorator will send a maximum of one parameter to the function it is decorating, which will be the message or content the client receives (in most cases).

Here's an example with the on decorator in use in a server. Here, the server has a command, `print_message_name`, and will print the message that it gets and who sent it.

```
server = ...

@server.on("print_message_name")
def on_print_message_name(client_data, message: str):
    print(f'{client_data.name} sent "{message}"')

server.start()
```

Here's another example with receiving data, this time on the client-side. The client will receive a command, `greet`, with a name. It will then print out a greeting with the name.

```
client = ...

@client.on("greet")
def on_greet(name: str):
    print(f"Hello there, {name}!")

client.start()
```

If the `threaded` parameter for the on decorator is True, then the function being decorated will run in a separate thread. This allows blocking code to run while still listening for updates.

It is useful if you want to get user input but also want to have the user receive other data.

```
client = ...

@client.on("ask_question", threaded=True)
def on_ask_question(question: str):
    """Contains blocking code with ``input()``."""
    answer = input(f"Please answer this question: {question}\n>")
    # ... send answer to server ...

@client.on("important")
def on_important(message: str):
    """This is important and cannot be missed!"""
    ...

client.start()
```

### Sending data

`HiSock` has multiple send methods. For now, we will be talking about sending to the server from one client or to one client from the server.

For the server: Sending data from the server to one client in `HiSock` uses the `send_client()` method. This method takes in a maximum of three parameters. The three parameters (in order) are a way to identify the client, the command to send, and the message being sent (optional). Although we won't be talking about it here, `send_all_clients()` does exactly what it says. It will do `send_client()` to all the clients that are connected, and only takes in the command and optional message

For the client: Sending data to the server in `HiSock` uses the `send()` method. This method takes a maximum of two parameters. The first parameter is the command to send, and the second parameter is the message being sent (optional).

Here is an example of sending data with a server-side code block:

```
server = ...

@server.on("join")
def on_client_join(client_data):
    server.send_client(client_data.ip, "ask_question", "Do you like sheep?")

@server.on("question_response")
def on_question_response(client_data, response: str):
    server.send_client(client_data.ip, "grade", 100)

server.start()
```

And here is an example on the client-side:

```
client = ...

@client.on("ask_question")
def on_ask_question(question: str):
    answer = input(f"Please answer this question: {question}\n>")
    client.send("question_response", answer)

@client.on("grade")
def on_grade(grade: int):
    print(f"You got a {grade:>3}%.")

client.start()
```

---

### Reserved events

As I stated before, not every receiver has a maximum of two parameters passed to it. Here are the cases where that is the case.

`HiSock` has reserved events. These events shouldn't be sent by the client or server explicitly as it is currently unsupported.

---

**Note:** Besides for `string` and `bytes` for `message`, these reserved events do not have type casting.

---

Here is a list of the reserved events:

Server:

- `join`

  The client sends the event `join` when they connect to the server. The only parameter sent to the function being decorated is the client data.

- `leave`

  The client sends the event `leave` when they disconnect from the server. The only parameter sent to the function being decorated is the client data.

- `name_change`

  The client sends the event `name_change` when they change their name. The parameters sent to the listening function are (in order) the client data, the old name, and the new name.

- `group_change`

  The client sends the event `group_change` when they change their group. The parameters sent to the listening function are (in order) the client data, the old group, and the new group.

- `message`

  When the server receives a command, it'll send an event to itself called `message` which will have two parameters. The two parameters are the client data who sent it and the raw data which was received.

- `*`

  This will be called when there is no listener for an incoming command and data. The three parameters are the client data, the command, and the content.

Client:

- `client_connect`

  When a client connects to the server, all the clients will have this event called. The only parameter for this is the client data for the client which joined.

- `client_disconnect`

  When a client disconnects from the server, all the clients will have this event called. The only parameter for this is the client data for the client which left.

- `force_disconnect`

  The server sends the event `force_disconnect` to a client when they kick the client. There are *no* parameters sent with the function that is being decorated with this.

- `*`

  This will be called when there is no listener for an incoming command and data. The two parameters are the command and the content.

### Type-casting

`HiSock` has a system called "type-casting" when transmitting data.

Data sent and received can be one of the following types:

- `bytes`
- `str`
- `int`
- `float`
- `bool`
- `None`
- `list` (with the types listed here)
- `dict` (with the types listed here)

---

**Note:** There is a type hint in `hisock.utils` called `Sendable` which has these.

---

The type that the data gets type-casted to depends on the type hint for the message argument for the function for the event receiving the data. If there is no type hint for the argument, the data received will be bytes.

Here are a few examples this server-side code block:

```
@server.on("string_sent")
def on_string_sent(client_data, message: str):
    """``message`` will be of type ``string``"""
    ...

@server.on("integer_sent")
def on_integer_sent(client_data, integer: int):
    """``integer`` will be of type ``int``"""
    ...

@server.on("dictionary_sent")
def on_dictionary_sent(client_data, dictionary: dict):
    """``dictionary`` will be of type ``dict``"""
    ...
```

---

**Note:** Although these examples are on the server-side, they work the exact same for the client-side.

---

Of course, you need to be careful that the type-casting will work. Turning `b"hello there"` to `int` will fail.

---

### Dynamic arguments

Remember where I said the on decorator will call the function with a *maximum* number of parameters?

In HiSock with an _unreserved_ event, the function to handle it can be called with the maximum number of parameters *or less*. Note that in a reserved event, dynamic arguments doesn't apply.

As an example, for the server: If an event has 1 argument, it will only be called with the client data. If it has 2 arguments, it will be called with the client data and the message. If it has 0 arguments, it'll be called as a void (no arguments).

Data can be sent similarly. If there is no data sent, the server will receive the equivalent of None for the type-casted data.

Here are a few examples of this with a server-side code block.

```
@server.on("event1")
def on_event1(client_data, message: str):
    print(f"I have {client_data=} and {message=} as a string!")

@server.on("event2")
def on_event2(client_data, message: int):
    print(f"I have {client_data=} and {message=} as an integer! {message+1=}")

@server.on("event3")
def on_event3(client_data):
    print(f"I only have {client_data=}!")

@server.on("event4")
def on_event4():
    print("I have nothing.")
```

Here is an example with a client-side code block that ties into the server-side code block above:

```
client.send("event1", "Hello")  # Server will receive "Hello"
client.send("event1")  # Server will receive an empty string
client.send("event2", b"123")  # Server will receive 123 and output 124
client.send("event2")  # Server will receive 0 and output 1
client.send("event3", "there")  # Server won't receive "there"
client.send("event4", "Hi")  # Server won't receive anything
```

---

### Conclusion

Now, you know how to:

- Create a server
- Create a client
- Transmit data
- Work with dynamic arguments
- Handle datatypes transmitted
- Do stuff with the data

Have fun HiSock-ing!

### 1.2.2 Intermediate-level Tutorial

**Table of Contents**



#### Before you start

It is imperative that you have basic knowledge of `HiSock`. You should know everything that the beginner tutorial focuses on. If you don't know, *read that now*!

This tutorial will focus on:

- Overriding reserved events
- How commands and data are sent
- Threading
- The other send methods
- The other receive methods
- Catch-all listeners (wildcard)
- Groups, names, and client info

---

#### Override

In `HiSock`, there are some reserved events. However, some use-cases may want to use the same name as the reserved events for unreserved events. In order to do this, you can override the reserved events. The `on` decorator has an argument for overriding a command. When set to `True`, the event will be treated like an unreserved event.

> **Warning:** This isn't recommended doing most times. Typically, the reserved events are used for handling clients. For example, if you override the join event, you'll have to have the client send a join event.

Here is an example with a server-side code-block.

```
server = ...

@server.on("leave", override=True)
def on_leave(client_data, reason: str):
    print(f"{client_data.name} left because {reason}")
    server.disconnect_client(client_data.ip, force=True)

server.run()
```

Now, the `leave` event won't be called when a client leaves. Instead, a client will manually send the `leave` event.

---

### Threading

In `HiSock`, there is an option for the client or server to run entirely in a different thread. This can be useful if you want to have a server that doesn't block the main thread, such as in conjunction with Pygame or Tkinter.

This is really simple to use! In place of `HiSockServer` or `HiSockClient`, you can use `ThreadedHiSockServer` or `ThreadedHiSockClient` respectively. You can also use `threaded_connect()` in place of `connect()` and `start_threaded_server()` in place of `start_server()`.

---

### How commands and data are sent

Commands and data are sent in a special syntax.

---

**Note:** In this section, text in <> is a placeholder for data.

---

### Reserved commands

For reserved commands, the syntax is different for each one.

For the client:

- `$KEEPALIVE$`

  This command is sent to the client from the server to make sure that the client is still connected.

- `$DISCONN$`

  This command is sent to the client from the server to disconnect the client.

- `$CLTCONN$<client info as a stringified dict>`

  This command is sent to the client from the server to inform that a new client connected.

- `$CLTDISCONN$<client info as a stringified dict>`

  This command is sent to the client from the server to inform that a client disconnected.

For the server:

- `<connecting socket is same as server socket>`

This happens when a new client connects to server.

- <bad client file number> OR <client data is falsy> OR $USRCLOSE$

  This happens when the client closes the connection and emits its leave, or it encounters an error when transmitting data. The client will be disconnected.

- $KEEPACK$

  This is sent to the client from the server to acknowledge that the client is still connected.

- $GETCLT$<client_identifier, either a name or stringified IP>

  This is sent to the server from the client to get the client's data.

- $CHNAME$<new name> OR $CHGROUP$<new group>

  This is sent to the server from the client to change the client's name or group, respectively.

---

### Unreserved commands

For reserved commands, the data is sent as follows:

- command and message sent

  $CMD$<command>$MSG$<message>

- command sent

  $CMD$<command>

---

### The other send methods

In HiSock, there are multiple send methods for the *server*. These methods are:

- send_client - sends a command and/or message to a singe client
- send_all_clients - sends a command and/or message to every client connected
- send_group - sends a command and/or message to every client in a group

There are also a few internal send methods that shouldn't need to be used. They are used for sending *raw* data. These methods are:

- _send_client_raw
- _send_all_clients_raw
- _send_group_raw

---

### A new way to receive data

In HiSock, there is also a different way to receive data.

Say you want to send data and wait for a response. Normally, you'd have to do something like this:

```
client = ...

@client.on("start")
def on_start():
    client.send(input("What would you like to say?"))
    print("Waiting for a response...")

@client.on("response")
def on_response(response: str):
    print(f"The server said: {response}")

client.start()
```

However, there is a better way! There is a method called `recv()`. This method has two parameters. The parameters (in order) are the command to receive on (optional) and the type to receive as (defaults to bytes).

The `recv()` method works like the `on` decorator. If the command to receive on is not specified, `recv()` will receive on any command or data that is sent and not caught by a function. Otherwise, it will only receive on the command. Then, the message received will be type-casted to the type specified and returned.

`recv()` is blocking, so the code in the function will pause until it's done. This is why it's recommended to use it in a threaded function.

Now, let's use the example above, but using the `recv()` method!

```
client = ...

@client.on("start", threaded=True)
def on_start():
    client.send(input("What would you like to say?"))
    print("Waiting for a response...")
    response = client.recv("response", str)
    print(f"The server said: {response}")

client.start()
```

---

### Catch-all listeners (wildcard)

In HiSock, there is a way to catch every piece of data sent that hasn't been handled by a listener already. This amazing thing is known as the catch-all or wildcard listener. Despite the name, this is not like the `message` reserved listener. It will only be called when there is no other handler for the data.

The function that will be called for the catch-all listener will be passed in the client data if it's a server, the command, and the message. If there is no message, it'll be type-casted into the equivalent of `None`.

Here is an example with a client-side and server-side code block:

```
client = ...

client.send(
   "hello i am an uncaught command",
   f"Random data: "
   + "".join(
      [
            chr(choice((randint(65, 90), randint(97, 122))))
            for _ in range(100)
      ]
   ),
)

@client.on("client_connect")
def on_connect(client_data):
    ...

@client.on("client_disconnect")
def on_disconnect(client_data):
    ...

@client.on("*")
def on_wildcard(command: str, data: str):
    print(f"The server sent some uncaught data: {command=}, {data=}")

client.start()
```

```
server = ...

@server.on("join")
def on_join(client_data):
   ...

@server.on("leave")
def on_leave(client_data):
   ...

@server.on("*")
def on_wildcard(client_data, command: str, data: str):
    print(
        f"There was some unhandled data from {client_data.name}. "
        f"{command=}, {data=}"
    )

    server.send_client(client_data, "i am also an uncaught command", data.replace("a", "
↪"))

server.start()
```

### Groups, name, and client info

In `HiSock`, each client has its own client data. Like mentioned in the previous tutorial, this client data can be a dictionary or an instance of `ClientInfo`.

Client info contains the following:

- `name`

  The name of the client. Will be `None` if not entered.

- `group`

  The group of the client. Will be `None` if not entered.

- `ip`

  The IP and port of the client as a string.

`ClientInfo` can also act like a dictionary, so you can access the client's data like this:

```
server = ...

@server.on("join")
def on_join(client_data):
    ip = client_data.ip  # Normal access
    name = client_data["name"]  # Dictionary-like access
    group = client_data.group   #Normal access
```

## 1.3 API Reference

These pages provide you with information about the functions available in `hisock`.

### 1.3.1 hisock.server

A module containing the main server classes and functions, including `HiSockServer` and `start_server()`

---

**Note:** Header lengths shouldn't be too long. By default, the header length is 16 bytes. This is good enough for most applications, as allows for 10**16 bytes to be sent.

---

**class** hisock.server.**HiSockServer**(*addr: tuple[str, int]*, *max_connections: int = 0*, *header_len: int = 16*, *cache_size: int = -1*, *keepalive: bool = True*)

The server class for `HiSock`.

**Parameters**

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number of where the server should be hosted. Due to the nature of reserved ports, it is recommended to host the server with a port number that's greater than or equal to 1024. **Only IPv4 is currently supported.**
- **max_connections** (*int, optional*) – The number of maximum connections the server should accept before refusing client connections. Pass in 0 for unlimited connections. Default passed in by *start_server()* is 0.

- **header_len** (*int, optional*) – An integer, defining the header length of every message. A larger header length would mean a larger maximum message length (about 10**header_len). Any client connecting **MUST** have the same header length as the server, or else it will crash. Default passed in by *start_server()* is 16 (maximum length: 10 quadrillion bytes).
- **cache_size** (*int, optional*) – The size of the message cache. -1 or below for no message cache, 0 for an unlimited cache size, and any other number for the cache size.
- **keepalive** (*bool, optional*) – A bool indicating whether a keepalive signal should be sent or not. If this is True, then a signal will be sent to every client every minute to prevent hanging clients in the server. The clients have thirty seconds to send back an acknowledge signal to show that they are still alive. Default is True.

**Variables**

- **addr** (*tuple*) – A two-element tuple containing the IP address and the port.
- **header_len** (*int*) – An integer storing the header length of each "message".
- **clients** (*dict*) – A dictionary with the socket as its key and the client info as its value.
- **clients_rev** (*dict*) – A dictionary with the client info as its key and the socket as its value (for reverse lookup, up-to-date with `clients`).
- **funcs** (*dict*) – A list of functions registered with decorator *on()*. **This is mainly used for under-the-hood-code.**

**Raises**

**TypeError** – If the address is not a tuple.

`close()`

Closes the server; ALL clients will be disconnected, then the server socket will be closed.

Running *server.run()* won't do anything now.

`disconnect_all_clients`(*force=False*)

Disconnect all clients.

`disconnect_client`(*client: Union[str, Tuple[str, int], ClientInfo], force: bool = False, call_func: bool = False*)

Disconnects a specific client.

**Parameters**

- **client** (*Client*) – The client to send data to. The format could be either by IP+port, a client name, or a `ClientInfo` instance.
- **force** (*bool, optional*) – A boolean, specifying whether to force a disconnection or not. Defaults to False.
- **call_func** – A boolean, specifying whether to call the `leave` reserved function when client is disconnected. Defaults to False.

**Raises**

- **ValueError** – If the client format is wrong.
- **ClientNotFound** – If the client does not exist.
- **UserWarning** – Using client name, and more than one client with the same name is detected.

`get_addr()` → tuple[str, int]

Gets the address of where the HiSock server is serving at.

**Returns**
A tuple of the address in the form of (ip, port)

**Return type**
tuple[str, int]

`get_all_clients`(*key: Union[Callable, str] = None*) → list[dict[str, str]]

Get all clients currently connected to the server. This is recommended over the class attribute `self._clients` or `self.clients_rev`, as it is in a dictionary-like format.

**Parameters**
**key** (*Union[Callable, str], optional*) – If specified, there are two outcomes: If it is a string, it will search for the dictionary for the key, and output it to a list (currently supports "ip", "name", "group"). If it is a callable, it will try to integrate the callable into the output with the `filter()` function.

**Returns**
A list of dictionaries, with the clients

**Return type**
list[dict, ...]

`get_client`(*client: Union[str, tuple[str, int]]*) → dict[str, Union[str, socket.socket]]

Gets the client data for a client from a name or tuple in the form of (ip, port).

**Returns**
The client data without the socket.

**Return type**
dict

**Raises**

- **ValueError** – Client format is wrong.
- **ClientNotFound** – Client does not exist.
- **UserWarning** – Using client name, and more than one client with the same name is detected.

`get_group`(*group: str*) → list[dict[str, Union[str, socket.socket]]]

Gets all clients from a specific group.

---

**Note:** If you want to get them from `clients_rev` directly, use `_get_all_client_sockets_in_group()` instead.

---

**Parameters**
**group** (*str*) – A string, representing the group to look up

**Raises**
**GroupNotFound** – Group does not exist

**Returns**
A list of dictionaries of clients in that group, containing the address, name, group, and socket

**Return type**
list

**on**(*command: str*, *threaded: bool = False*, *override: bool = False*) → Callable

A decorator that adds a function that gets called when the server receives a matching command.

Reserved functions are functions that get activated on specific events, and they are:

1. `join` - Activated when a client connects to the server
2. `leave` - Activated when a client disconnects from the server
3. `message` - Activated when a client messages to the server
4. `name_change` - Activated when a client changes its name
5. `group_change` - Activated when a client changes its group

The parameters of the function depend on the command to listen. For example, reserved commands `join` and `leave` have only one client parameter passed, while reserved command `message` has two: client data and message. Other unreserved functions will also be passed in the same parameters as `message`.

In addition, certain type casting is available to both reserved and unreserved functions. That means, that, using type hints, you can automatically convert between needed instances. The type casting currently supports:

- `bytes`
- `str`
- `int`
- `float`
- `bool`
- `None`
- `list` (with the types listed here)
- `dict` (with the types listed here)

For more information, read the documentation for type casting.

**Parameters**

- **command** (*str*) – A string, representing the command the function should activate when receiving it.
- **threaded** (*bool, optional*) – A boolean, representing if the function should be run in a thread in order to not block the run loop. Default is False.
- **override** (*bool, optional*) – A boolean representing if the function should override the reserved function with the same name and to treat it as an unreserved function. Default is False.

**Returns**
The same function (the decorator just appended the function to a stack).

**Return type**
function

**Raises**
**TypeError** – If the number of function arguments is invalid.

**send_all_clients**(*command: str*, *content: Union[bytes, str, int, float, None,* ClientInfo*, List[Union[str, int, float, bool, None, dict, list]], Dict[str, Union[str, int, float, bool, None, dict, list]]] = None*)

Sends the command and content to *ALL* clients connected.

**Parameters**

- **command** (*str*) – A string, representing the command to send to every client.
- **content** (*Sendable, optional*) – The message / content to send

**send_client**(*client: Union[str, Tuple[str, int], ClientInfo], command: str, content: Union[bytes, str, int, float, None, ClientInfo, List[Union[str, int, float, bool, None, dict, list]], Dict[str, Union[str, int, float, bool, None, dict, list]]] = None*)

Sends data to a specific client.

**Parameters**

- **client** (*Client*) – The client to send data to. The format could be either by IP+port, or a client name.
- **command** (*str*) – A string, containing the command to send.
- **content** (*Sendable*) – The message / content to send

**Raises**

- **ValueError** – Client format is wrong.
- **ClientNotFound** – Client does not exist.
- **UserWarning** – Using client name, and more than one client with the same name is detected.

**send_group**(*group: Union[str, ClientInfo], command: str, content: Union[bytes, str, int, float, None, ClientInfo, List[Union[str, int, float, bool, None, dict, list]], Dict[str, Union[str, int, float, bool, None, dict, list]]] = None*)

Sends data to a specific group. Groups are recommended for more complicated servers or multipurpose servers, as it allows clients to be divided, which allows clients to be sent different data for different purposes.

**Parameters**

- **group** (*Union[str, ClientInfo]*) – A string or a ClientInfo, representing the group to send data to. If the group is a ClientInfo, and the client is in a group, the method will send data to that group.
- **command** (*str*) – A string, containing the command to send
- **content** (*Union[bytes, dict]*) – A bytes-like object, with the content/message to send

**Raises**

**TypeError** – If the group does not exist, or the client is not in a group (ClientInfo).

**start**(*callback: Optional[Callable] = None, error_handler: Optional[Callable] = None*)

Start the main loop for the server.

**Parameters**

- **callback** (*Callable, optional*) – A function that will be called every time the client receives and handles a message.
- **error_handler** (*Callable, optional*) – A function that will be called every time the client encounters an error.

**class** hisock.server.**ThreadedHiSockServer**(**args*, ***kwargs*)

HiSockClient, but running in its own thread as to not block the main loop. Please note that while this is running in its own thread, the event handlers will still be running in the main thread. To avoid this, use the threaded=True argument for the on decorator.

For documentation purposes, see HiSockClient.

**start**(*callback: Optional[Callable] = None*, *error_handler: Optional[Callable] = None*)

Starts the main server loop. For documentation, see *HiSockServer.start()*.

**close**()

Closes the server. Blocks the thread until the server is closed. For documentation, see *HiSockServer.close()*.

---

hisock.server.**start_server**(*addr*, *max_connections=0*, *header_len=16*)

Creates a *HiSockServer* instance. See *HiSockServer* for more details and documentation.

**Returns**

A *HiSockServer* instance.

hisock.server.**start_threaded_server**(**args*, ***kwargs*)

Creates a *ThreadedHiSockServer* instance. See *ThreadedHiSockServer* for more details. For documentation, see *start_server()*.

**Returns**

A *ThreadedHiSockServer* instance

### 1.3.2 hisock.client

A module containing the main client classes and functions, including HiSockClient and connect()

**class** hisock.client.**HiSockClient**(*addr: tuple[str, int]*, *name: Union[str, None] = None*, *group: Union[str, None] = None*, *header_len: int = 16*, *cache_size: int = -1*)

The client class for HiSock.

**Parameters**

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number of where the server is hosted. **Only IPv4 is currently supported.**
- **name** (*str, optional*) – Either a string or NoneType, representing the name the client goes by. Having a name provides an easy interface of sending. data to a specific client and identifying clients. It is therefore highly recommended to pass in a name.

  Pass in NoneType for no name (*connect()* should handle that)
- **group** (*str, optional*) – Either a string or NoneType representing the group the client is in. Being in a group provides an easy interface of sending data to multiple specific clients, and identifying multiple clients. It is highly recommended to provide a group for complex servers. Pass in NoneType for no group (*connect()* should handle that).
- **header_len** (*int, optional*) – An integer defining the header length of every message. A larger header length would mean a larger maximum message length (about 10**header_len). **MUST** be the same header length as the server, or else it will crash (hard to debug too!). Default sets to 16 (maximum length of content: 10 quadrillion bytes).
- **cache_size** (*int, optional*) – The size of the message cache. -1 or below for no message cache, 0 for an unlimited cache size, and any other number for the cache size.

**Variables**

- **addr** (*tuple*) – A two-element tuple containing the IP address and the port number of the server.
- **header_len** (*int*) – An integer storing the header length of each "message".
- **name** (*str*) – A string representing the name of the client to identify by. Default is None.
- **group** (*str*) – A string representing the group of the client to identify by. Default is None.
- **funcs** (*dict*) – A list of functions registered with decorator *on()*. **This is mainly used for under-the-hood-code.**
- **connect_time** (*int*) – An integer sotring the Unix timestamp of when the client connected to the server.

**change_group**(*new_group: Optional[str]*)

Changes the client's group.

**Parameters**

**new_group** (*Union[str, None]*) – The new group name for the client to be called. if left blank, then the group will be reset

**change_name**(*new_name: Optional[str]*)

Changes the name of the client

**Parameters**

**new_name** (*Union[str, None]*) – The new name for the client to be called. If left blank, then the name will be reset.

**close**(*emit_leave: bool = True*)

Closes the client; running `client.update()` won't do anything now

**Parameters**

**emit_leave** (*bool*) – Decides if the client will emit *leave* to the server or not

**get_cache**(*idx: Union[int, slice, None] = None*) → list[*MessageCacheMember*]

Gets the message cache.

**Parameters**

**idx** (*Union[int, slice], optional*) – An integer or `slice`, specifying what specific message caches to return. Default is None (retrieves the entire cache).

**Returns**

A list of dictionaries, representing the cache

**Return type**

list[dict]

**get_client**(*client: Union[str, Tuple[str, int],* ClientInfo*], get_as_dict: bool = False*) → Union[*ClientInfo*, dict]

Gets the client data for a client.

**Parameters**

- **client** (*Client*) – The client name or IP+port to get.
- **get_as_dict** (*bool, optional*) – A boolean representing if the client data should be returned as a dictionary. Otherwise, it'll be returned as an instance of `ClientInfo`. Default is False.

**Returns**
The client data.

**Return type**
Union[*ClientInfo*, dict]

**Raises**

- **ValueError** – If the client IP is invalid.
- **ClientNotFound** – If the client couldn't be found.
- **ServerException** – If another error occurred.

**get_client_addr**() → tuple[str, int]

Gets the address of the client.

**Returns**
A tuple, with the format (IP, port).

**Return type**
tuple[str, int]

**get_server_addr**() → tuple[str, int]

Gets the address of where the client is connected to.

**Returns**
A tuple, with the format (str IP, int port)

**Return type**
tuple[str, int]

**on**(*command: str*, *threaded: bool = False*, *override: bool = False*) → Callable

A decorator that adds a function that gets called when the client receives a matching command.

Reserved functions are functions that get activated on specific events, and they are:

1. `client_connect` - Activated when a client connects to the server
2. `client_disconnect` - Activated when a client disconnects from the server

The parameters of the function depend on the command to listen. For example, reserved functions `client_connect` and `client_disconnect` gets the client's data passed in as an argument. All other unreserved functions get the message passed in.

In addition, certain type casting is available to unreserved functions. That means, that, using type hints, you can automatically convert between needed instances. The type casting currently supports:

- `bytes`
- `str`
- `int`
- `float`
- `bool`
- `None`
- `list` (with the types listed here)
- `dict` (with the types listed here)

For more information, read the documentation for type casting.

**Parameters**

- **command** (*str*) – A string, representing the command the function should activate when receiving it.
- **threaded** (*bool, optional*) – A boolean, representing if the function should be run in a thread in order to not block the update loop. Default is False.
- **override** (*bool, optional*) – A boolean representing if the function should override the reserved function with the same name and to treat it as an unreserved function. Default is False.

**Returns**
The same function (the decorator just appended the function to a stack).

**Return type**
function

**Raises**
**TypeError** – If the number of function arguments is invalid.

**send**(*command: str*, *content: Union[bytes, str, int, float, None,* ClientInfo*, List[Union[str, int, float, bool, None, dict, list]], Dict[str, Union[str, int, float, bool, None, dict, list]]] = None*)

Sends a command & content to the server.

**Parameters**

- **command** (*str*) – A string, containing the command to send
- **content** (*Sendable, optional*) – The message / content to send

**start**(*callback: Optional[Callable] = None*, *error_handler: Optional[Callable] = None*)

Start the main loop for the client.

**Parameters**

- **callback** (*Callable, optional*) – A function that will be called every time the client receives and handles a message.
- **error_handler** (*Callable, optional*) – A function that will be called every time the client encounters an error.

---

class hisock.client.**ThreadedHiSockClient**(**args*, ***kwargs*)

*HiSockClient*, but running in its own thread as to not block the main loop. Please note that while this is running in its own thread, the event handlers will still be running in the main thread. To avoid this, use the `threaded=True` argument for the `on` decorator.

For documentation purposes, see *HiSockClient*.

**start**(*callback: Optional[Callable] = None*, *error_handler: Optional[Callable] = None*)

Starts the main client loop. For documentation, see *HiSockClient.start()*.

**close**(**args*, ***kwargs*)

Closes the client. Blocks the thread until the client is closed. For documentation, see *HiSockClient.close()*.

---

hisock.client.**connect**(*addr*, *name=None*, *group=None*, *header_len=16*, *cache_size=-1*)

Creates a *HiSockClient* instance. See HiSockClient for more details

**Parameters**

- **addr** (*tuple*) – A two-element tuple containing the IP address and the port number of the server.
- **name** (*str, optional*) – A string containing the name of what the client should go by. This argument is optional.
- **group** (*str, optional*) – A string, containing the "group" the client is in. Groups can be utilized to send specific messages to them only. This argument is optional.
- **header_len** (*int, optional*) – An integer defining the header length of every message. Default is True.
- **cache_size** (*int, optional*) – The size of the message cache. -1 or below for no message cache, 0 for an unlimited cache size, and any other number for the cache size.

**Returns**
A *HiSockClient* instance.

**Return type**
instance

---

**Note:** A simple way to use this function is to use `utils.input_client_config()` which will ask you for the server IP, port, name, and group. Then, you can call this function by simply doing `connect(*input_client_config())`

---

hisock.client.**threaded_connect**(**args*, ***kwargs*)

Creates a *ThreadedHiSockClient* instance. See *ThreadedHiSockClient* for more details :return: A *ThreadedHiSockClient* instance

### 1.3.3 hisock.utils

A module containing some utilities to either:

1. Help `hisock.client` and `hisock.server` run (denoted with leading underscore) 1. Provide functions to use alongside `HiSock`

**class** hisock.utils.**MessageCacheMember**(*message_dict: dict*)

**class** hisock.utils.**ClientInfo**(*ip*, *name*, *group*)

hisock.utils.**make_header**(*header_message: Union[str, bytes]*, *header_len: int*, *encode=True*) → Union[str, bytes]

Makes a header of `header_message`, with a maximum header length of `header_len`

**Parameters**

- **header_message** (*Union[str, bytes]*) – A string OR bytes-like object, representing the data to make a header from
- **header_len** (*int*) – An integer, specifying the actual header length (will be padded)
- **encode** – A boolean, specifying the

**Returns**
The constructed header, padded to `header_len` bytes

**Return type**
Union[str, bytes]

hisock.utils.**receive_message**(*connection: socket.socket*, *header_len: int*) → Union[dict[str, bytes], bool]

Receives a message from a server or client.

**Parameters**

- **connection** (*socket.socket*) – The socket to listen to for messages. **MUST BE A SOCKET**
- **header_len** (*int*) – The length of the header, so that it can successfully retrieve data without loss/gain of data

**Returns**

A dictionary, with two key-value pairs; The first key-value pair refers to the header, while the second one refers to the actual data

**Return type**

Union[dict[“header”: bytes, “data”: bytes], False

hisock.utils.**validate_command_not_reserved**(*command: str*)

Checks for illegal $cmd$ notation (used for reserved functions).

**Parameters**

**command** (*str*) – The command to check.

**Raises**

**ValueError** – If the command is reserved.

hisock.utils.**validate_ipv4**(*ip: Union[str, tuple]*, *require_ip: bool = True*, *require_port: bool = True*) → bool

Validates an IPv4 address. If the address isn't valid, it will raise an exception. Otherwise, it'll return True

**Parameters**

- **ip** (*Union[str, tuple]*) – The IPv4 address to validate.
- **require_ip** (*bool*) – Whether or not to require an IP address. If True, it will raise an exception if no IP address is given. If False, this will only check the port.
- **require_port** (*bool*) – Whether or not to require a port to be specified. If True, it will raise an exception if no port is specified. If False, it will return the address as a tuple without a port.

**Returns**

True if there were no exceptions.

**Return type**

Literal[“True”]

**Raises**

**ValueError** – IP address is not valid

hisock.utils.**get_local_ip**(*all_ips: bool = False*) → str

Gets the local IP of your device, with sockets

**Parameters**

**all_ips** (*bool, optional*) – A boolean, specifying to return all the local IPs or not. If set to False (the default), it will return the local IP first found by `socket.gethostbyname()`

Default: False

**Returns**

A string containing the IP address, in the format “ip:port”

**Return type**

str

hisock.utils.**input_server_config**(*ip_prompt: str = 'Enter the IP for the server: '*, *port_prompt: str = 'Enter the port for the server: '*) → tuple[str, int]

Provides a built-in way to obtain the IP and port of where the server should be hosted, through `input()`.

**Parameters**

- **ip_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for IP.
- **port_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for port

**Returns**
A two-element tuple, consisting of IP and port.

**Return type**
tuple[str, int]

hisock.utils.**input_client_config**(*ip_prompt: str = 'Enter the IP of the server: '*, *port_prompt: str = 'Enter the port of the server: '*, *name_prompt: Union[str, None] = 'Enter name: '*, *group_prompt: Union[str, None] = 'Enter group to connect to: '*) → tuple[tuple[str, int], Optional[str], Optional[str]]

Provides a built-in way to obtain the IP and port of the configuration of the server to connect to.

**Parameters**

- **ip_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for IP.
- **port_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for port.
- **name_prompt** (*Union[str, None], optional*) – A string, specifying the prompt to show when asking for client name.
- **group_prompt** (*Union[str, None], optional*) – A string, specifying the prompt to show when asking for client group.

**Returns**
A tuple containing the config options of the server. Will filter out the unused options.

**Return type**
tuple[str, int, Optional[str], Optional[int]]

hisock.utils.**ipstr_to_tup**(*formatted_ip: str*) → tuple[str, int]

Converts a string IP address into a tuple equivalent.

**Parameters**
**formatted_ip** (*str*) – A string, representing the IP address. Must be in the format "ip:port".

**Returns**
A tuple, with a string IP address as the first element and an integer port as the second element.

**Return type**
tuple[str, int]

**Raises**
**ValueError** – If the IP address isn't in the "ip:port" format.

hisock.utils.**iptup_to_str**(*formatted_tuple: tuple[str, int]*) → str

Converts a tuple IP address into a string equivalent. This function is the opposite of *ipstr_to_tup()*.

**Parameters**
**formatted_tuple** (*tuple[str, int]*) – A two-element tuple, containing the IP address and the port. Must be in the format (ip: str, port: int).

**Returns**
A string, with the format "ip:port".

**Return type**
str

**Raises**
**ValueError** – If the IP address isn't in the "ip:port" format.

## 1.4 Changelog

This page keeps track of all the new features that were added to, modified, or removed in specific versions. This may be useful for selecting which version is best for you, if the latest version does not work for you.

---

### 1.4.1 v1.1

#### New features

- A message cache for HiSockClient! The message cache allows you to view the last x messages sent by the client. Configuration of the message cache is available on creation of the instance.
- `.disconnect_client()`, `.disconnect_all_clients()`, and `.close()` methods for HiSockServer, as well as the `force_disconnect` reserved function for HiSockClient! It's now easier to force disconnect of a client from a server with these functions. The `force_disconnect` reserved function will be triggered when the client has been disconnected from the server with these functions.
- Add list and dict type casts for HiSockClient.

#### Improved features

- Documentation improvements, yay!
- Handle hisock errors better (not crashes!). New exceptions have been created for specific causes of server/client errors.
- cleancode.py can now clean up the code, with the help of black! So, the code has been PEP8-ified, for our (my) avid PEP8 fans.
- Move HiSockClient and HiSockServer to be available on import of hisock.
- Move __version__ to be available on import of hisock.
- Separate requirements.txt into that and requirements_contrib.txt. Some of the requirements are actually just requirements for the tests to run.

#### Bug fixes

- (Maybe) fix bug for the .close() method of HiSockClient.
- Fixed a fatal import error of hisock.

#### Other

Hisock now has a new logo, created by sheepy0125! Hisock also has a discord server! However, we're (I'm) still setting it up, so the invite isn't public yet.

---

### 1.4.2 v1.0

#### New features

- New examples have been added to hisock! Now, you can play a Tic-Tac-Toe game made in hisock! There is also the addition of the example shown in the README
- HiSockClient, HiSockServer, and their threaded counterparts now support some dunder methods!
- More type casts have been added
- Ability to change name and group after client initialization has been added (change_name() and change_group())
- A built-in way to obtain server and client configuration through inputs has been added (input_server_config() and input_client_config())

#### Improved features

- Of course, we have some documentation improvements!
- Pictures are starting to appear in the documentation
- I added a beginner's tutorial to get started on hisock
- I am currently working on another intermediate hisock tutorial, which covers the more advanced topics
- A new changelog page has been added
- Python docstrings have been improved
- Hisock error handling has been improved again
- Added classifiers in PyPI

#### Bug fixes

- The regular expression used in 0.1 and earlier has been replaced with a newer one, in order to respond against certain edge cases
- Bumped cryptography module from 3.4.8 to 35.0.0 (security patches)

---

### 1.4.3 v0.1

This version is the first minor version of `hisock`! It contains several major code accessibility things added, though not a lot of in-usage code has been added in this version.

#### New features

- PyPI installation of hisock (`python -m pip install hisock` or `pip3 install hisock`)
- Documentation for `hisock`, hosted here! (ReadTheDocs)
- New support for threaded `HiSockServer` and `HiSockClient`

#### Improved features

- Better traceback handling

---

### 1.4.4 v0.0.1 (GRAND RELEASE)

The first version of hisock! Contains all the basics of what hisock can do, including:

- Name/Group feature to identify specific clients
- Decorators to handle message receiving
- Under-the-hood code that handles headers
- Type-casting receiving function arguments

# INDEX