hisock

Release 0.0.2

SSS-Says-Snek

Sep 22, 2021

CONTENTS:

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!

Tutorials will be added eventually, but for now, there are only installation steps and some examples

1.1.1 Understanding hisock

First of all, what... **IS** hisock? Well, it's a higher-level extension of the `socket` module, with simpler and more efficient usages, sure, but for beginners, it may be a bit confusing.

What is the socket module?

I've mentioned a lot about a `socket` module in Python a lot, but some of y'all may not know *what* socket is. Basically, it's a pretty low-level networking interface that uses "sockets" to communicate between computers over a network. The problem is, it's a bit overwhelming when you start learning sockets.

So, I developed `hisock`, which basically simplifies `socket` down, and provides additional features.

What are the advantages of using hisock over socket?

That's a good question. While `hisock` is still under development, it aims to simplify or eliminate some complex parts of the standard `socket`. For example, hisock uses decorators to simplify code structure, and eliminates the hassle of worrying about headers.

Note: Again, some of you may not know what a header is. When you send data, it is not interpreted as a "message"; instead of messages, there is a "stream" of data, and the client/server decides how many bytes to read from the stream. This creates a problem; how do we know how much of a message to read? This is where headers come in. They are basically data the specifies the length of a "message". In order for this to work, headers **MUST** be fixed-length, so it is usually padded with spaces.

Let's say that I decided that my header length would be 16 bytes long. When a client sends me some data, it will have that header in front, then the actual content. I would first receive the first 16 bytes, and see that it is the number "12",

followed by 14 spaces. At this point, I know that the "message" is 12 bytes long. So, I receive another 12 bytes, to get the message "Hello World!"

1.1.2 Installation

There are two options when installing hisock:

1. You can download a zip of it on Github. It is located here. You can either clone it with the website or Github Desktop, but you can also clone it with Git:

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

Then, copy-paste the module into your code, and it should work!

> **Caution:** This method of installing is **heavily** discouraged, unless you have modified hisock enough. However, if you had lots of modified changes, you could open a pull request on Github, and wait for me to approve it

2. You can install it via pip (RECOMMENDED). You can use the following command in the terminal/command prompt:

```
$ python -m pip install hisock (Windows)
$ pip3 install hisock (Mac/Linux, I think)
```

Then, you have hisock installed to your python version! An extremely big plus of this installation over the manual installation is that you are able to use it anywhere on your computer, without cloning it into your directory.

> **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

1.1.3 Examples

Examples are located in the *./examples* directory in hisock. You can take a look at some of them. The basic client-server examples are:

Client

```
"""
Basic example of the structure of `hisock`. This is the client script.
Not an advanced example, but gets the main advantages of hisock across
"""

import time

from hisock import connect, get_local_ip

server_to_connect = input("Enter server IP to connect to (Press enter for default of␣
↪your local IP): ")
```

(continues on next page)

(continued from previous page)

```
port = input("Enter Server port number (Press enter for default of 6969): ")

if server_to_connect == '':
    server_to_connect = get_local_ip()

if port == '':
    port = 6969
else:
    port = int(port)

name = input("Name? (Press enter for no name) ")
group = input("Group? (Press enter for no group) ")

print("======================================== ESTABLISHING CONNECTION␣
↪========================================")

if name == '':
    name = None
if group == '':
    group = None

join_time = time.time()

s = connect(
    (server_to_connect, port),
    name=name, group=group)

@s.on("hello_message")
def handle_hello(msg):
    print("Thanks, server, for sending a hello, just for me!")
    print(f"Looks like, the message was sent on timestamp {msg.decode()}, "
          f"which is just {round(float(msg.decode()) - join_time, 6) * 1000}␣
↪milliseconds since the connection!")
    print("In response, I'm going to send the server a request to do some processing")

    s.send("processing1", b"randnum**2")
    result = int(s.recv_raw())

    print(f"WHOAAA! The result is {result}! Thanks server!")

while True:
    s.update()
```

Server

```
import sys
import time
import random

from hisock import start_server, get_local_ip, iptup_to_str

ADDR = get_local_ip()
```

(continues on next page)

(continued from previous page)

```
PORT = 6969

if len(sys.argv) == 2:
    ADDR = sys.argv[1]
if len(sys.argv) == 3:
    PORT = int(sys.argv[2])

print(f"Serving at {ADDR}")
server = start_server((ADDR, PORT))

@server.on("join")
def client_join(client_data):
    print(f"Cool, {':'.join(map(str, client_data['ip']))} joined!")
    if client_data['name'] is not None:
        print(f"    - With a sick name \"{client_data['name']}\", very cool!")
    if client_data['group'] is not None:
        print(f"    - In a sick group \"{client_data['group']}\", cool!")

    print("I'm gonna send them a quick hello message")

    server.send_client(client_data['ip'], "hello_message", str(time.time()).encode())


@server.on("processing1")
def process(client, process_request: str):
    print(f"\nAlright, looks like {iptup_to_str(client['ip'])} received the hello␣
↪message, "
          "\nas now they're trying to compute something on the server, because they have
↪"
          "potato computers")
    print("Their processing request is:", process_request)

    for _ in range(process_request.count("randnum")):
        randnum = str(random.randint(1, 100000000))
        process_request = process_request.replace("randnum", randnum, 1)

    result = eval(process_request)  # Insecure, but I'm lazy, so...
    print(f"Cool! The result is {result}! I'mma send it to the client")
    server.send_client_raw(client['ip'], str(result).encode())


while True:
    server.run()
```

1.2 API Reference

1.2.1 hisock.server

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

Note: Header lengths usually should not be that long; on average, header lengths are about 64 bytes long, which is more than enough for most cases (10 vigintillion, 10**64). Plus, a release is planned for hisock that utilizes ints to bump the header utilization from *10**x* to *2**(7x)* (where x is the header length)

---

class hisock.server.**HiSockServer**(*addr: tuple[str, int]*, *blocking: bool = True*, *max_connections: int = 0*, *header_len: int = 16*, *tls: Union[dict, str] = None*)

The server class for hisock HiSockServer offers a neater way to send and receive data than sockets. You don't need to worry about headers now, yay!

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 higher than 1023. Only IPv4 currently supported
- **blocking** (*bool, optional*) – A boolean, set to whether the server should block the loop while waiting for message or not. Default passed in by *start_server()* is True
- **max_connections** (*int, optional*) – The number of maximum connections *HiSockServer* should accept, before refusing clients' 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 smaller header length would mean a smaller 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)

Variables

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number
- **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
- **funcs** (*dict*) – A list of functions registered with decorator *on()*. **This is mainly used for under-the-hood-code**

Note: It is advised to use *get_client()* or *get_all_clients()* instead of using `clients` and `clients_rev`

---

get_addr()

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

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

`get_all_clients`(*key: Optional[Union[Callable, str]] = None*)
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 support "ip", "name", "group"). Finally, if it is a callable, it will try to integrate the callable into the output (CURRENTLY NOT SUPPORTED YET)

Returns A list of dictionaries, with the clients

Return type list[dict, ...]

`get_client`(*client: Union[str, tuple[str, int]]*)
Gets a specific client's information, based on either:

1. The client name
2. The client IP+Port
3. The client IP+Port, in a 2-element tuple

Parameters `client` (*Union[str, tuple]*) – A parameter, representing the specific client to look up. As shown above, it can either be represented as a string, or as a tuple.

Raises

- `ValueError` – Client argument is invalid
- `TypeError` – Client does not exist

Returns A dictionary of the client's info, including IP+Port, Name, Group, and Socket

Return type dict

`get_group`(*group: str*)
Gets all clients from a specific group

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

Raises `TypeError` – 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*)
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. Currently, there are 3 for HiSock-Server:

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

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 nonreserved functions will also be passed in the same parameters as *message*

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

1. bytes -> int (Will raise exception if bytes is not numerical)
2. bytes -> str (Will raise exception if there's a unicode error)

Type casting for reserved commands is scheduled to be implemented, and is currently being worked on.

Parameters **command** (*str*) – A string, representing the command the function should activate when receiving it

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

Return type function

run()

Runs the server. This method handles the sending and receiving of data, so it should be run once every iteration of a while loop, as to not lose valuable information

---

Note: This is the main method to run HiSockServer. This **MUST** be called every iteration in a while loop, as to keep waiting time as short as possible between client and server. It is also recommended to put this in a thread.

---

send_all_clients(*command: str*, *content: bytes*)

Sends the commmand and content to *ALL* clients connected

Parameters

- **command** (*str*) – A string, representing the command to send to every client
- **content** (*bytes*) – A bytes-like object, containing the message/content to send to each client

send_client(*client: Union[str, tuple]*, *command: str*, *content: bytes*)

Sends data to a specific client. Different formats of the client is supported. It can be:

- An IP + Port format, written as "ip:port"
- A client name, if it exists

Parameters

- **client** (*Union[str, tuple]*) – 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** (*bytes*) – A bytes-like object, with the content/message to send

Raises

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

send_client_raw(*client*, *content: bytes*)

Sends data to a specific client, *without a command* Different formats of the client is supported. It can be:

- An IP + Port format, written as "ip:port"
- A client name, if it exists

Parameters

- **client** (*Union[str, tuple]*) – The client to send data to. The format could be either by IP+Port, or a client name
- **content** (*bytes*) – A bytes-like object, with the content/message to send

Raises

- **ValueError** – Client format is wrong
- **TypeError** – Client does not exist
- **Warning** – Using client name and more than one client with the same name is detected

send_group(*group: str*, *command: str*, *content: bytes*)

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** (*str*) – A string, representing the group to send data to
- **command** (*str*) – A string, containing the command to send
- **content** (*bytes*) – A bytes-like object, with the content/message to send

Raises **TypeError** – The group does not exist

send_group_raw(*group: str*, *content: bytes*)

Sends data to a specific group, without commands. 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.

Non-command-attached content is recommended to be used alongside with `HiSockClient.recv_raw()`

Parameters

- **group** (*str*) – A string, representing the group to send data to
- **content** (*bytes*) – A bytes-like object, with the content/message to send

Raises **TypeError** – The group does not exist

class hisock.server.**ThreadedHiSockServer**(*addr*, *blocking=True*, *max_connections=0*, *header_len=16*)

A downside of *HiSockServer* is that you need to constantly *run()* it

join()

Waits for the thread to be killed

run()

The main while loop to run the thread

Refer to *HiSockServer* for more details

> **Warning:** This method is **NOT** recommended to be used in an actual production enviroment. This is used internally for the thread, and should not be interacted with the user

start_server()

Starts the main server loop

stop_server()

Stops the server

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

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

Returns A *HiSockServer* instance

hisock.server.**start_threaded_server**(*addr*, *blocking=True*, *max_connections=0*, *header_len=16*)
Creates a *ThreadedHiSockServer* instance. See *ThreadedHiSockServer* for more details

Returns A *ThreadedHiSockServer* instance

1.2.2 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]*, *group: Union[str, None]*, *blocking: bool = True*, *header_len: int = 16*)
The client class for hisock. HiSockClient offers a higher-level version of sockets. No need to worry about headers now, yay! HiSockClient also utilizes decorators to receive messages, as an easy way of organizing your code structure (methods are provided, like *recv_raw()*, of course)

Parameters

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number of the server wishing to connect to Only IPv4 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)
- **blocking** (*bool, optional*) – A boolean, set to whether the client should block the loop while waiting for message or not. Default sets to True
- **header_len** (*int, optional*) – An integer, defining the header length of every message. A smaller header length would mean a smaller maximum message length (about 10**header_len). The header length MUST be the same as the server connecting, or it will crash (hard to debug though). Default sets to 16 (maximum length of content: 10 quadrillion bytes)

Variables

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number
- **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. Defaults to None
- **group** (*str*) – A string, representing the group of the client to identify by. Defaults to None
- **funcs** (*dict*) – A list of functions registered with decorator *on()*.

  This is mainly used for under-the-hood-code

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

on(*command: str*)
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. Currently, there are 2 for HiSockClient:

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 nonreserved functions get the message passed in.

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

1. bytes -> int (Will raise exception if bytes is not numerical)
2. bytes -> str (Will raise exception if there's a unicode error)

Type casting for reserved commands is scheduled to be implemented, and is currently being worked on.

Parameters **command** (*str*) – A string, representing the command the function should activate when receiving it

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

Return type function

raw_send(*content: bytes*)
Sends a message to the server: NO COMMAND REQUIRED. This is preferable in some situations, where clients need to send multiple data over the server, without overcomplicating it with commands

Parameters **content** (*bytes*) – A bytes-like object, with the content/message to send

recv_raw()
Waits (blocks) until a message is sent, and returns that message. This is not recommended for content with commands attached; it is meant to be used alongside with `HiSockServer.send_client_raw()` and `HiSockServer.send_group_raw()`

Returns A bytes-like object, containing the content/message the client first receives

Return type bytes

send(*command: str*, *content: bytes*)
Sends a command & content to the server, where it will be interpreted

Parameters

- **command** (*str*) – A string, containing the command to send
- **content** (*bytes*) – A bytes-like object, with the content/message to send

update()
Handles newly received messages, excluding the received messages for *wait_recv* This method must be called every iteration of a while loop, as to not lose valuable info. In some cases, it is recommended to run this in a thread, as to not block the program

---

Note: This is the main method to run HiSockClient. This **MUST** be called every iteration in a while loop, as to keep waiting time as short as possible between client and server. It is also recommended to put this in a thread.

---

class hisock.client.**ThreadedHiSockClient**(*addr*, *name=None*, *group=None*, *blocking=True*, *header_len=16*)

A downside of `HiSockServer` is that you need to constantly *run()* it

join()

Waits for the thread to be killed

run()

The main while loop to run the thread

Refer to *HiSockClient* for more details

> **Warning:** This method is **NOT** recommended to be used in an actual production enviroment. This is used internally for the thread, and should not be interacted with the user

start_client()

Starts the main server loop

stop_client()

Stops the client

hisock.client.**connect**(*addr*, *name=None*, *group=None*, *blocking=True*, *header_len=16*)

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

Parameters

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number
- **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
- **blocking** (*bool, optional*) – A boolean, specifying if the client should block or not in the socket.

  Defaults to True
- **header_len** (*int, optional*) – An integer, defining the header length of every message.

  Defaults to True

Returns A *HiSockClient* instance

Return type instance

hisock.client.**threaded_connect**(*addr*, *name=None*, *group=None*, *blocking=True*, *header_len=16*)

Creates a `ThreadedHiSockServer` instance. See `ThreadedHiSockServer` for more details

Returns A `ThreadedHiSockServer` instance

INDEX