

---

# **Telethon**

***Release 2.0.0a0***

**Lonami**

**Nov 29, 2023**



# FIRST STEPS

<b>1</b>	<b>First steps</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Signing in . . . . .	4
1.3	Next steps . . . . .	8
<b>2</b>	<b>Concepts</b>	<b>9</b>
2.1	Chats . . . . .	9
2.2	Updates . . . . .	11
2.3	Messages . . . . .	14
2.4	Sessions . . . . .	19
2.5	RPC Errors . . . . .	20
2.6	HTTP Bot API vs MTProto . . . . .	20
2.7	The Full API . . . . .	27
2.8	Data centers . . . . .	28
2.9	Glossary . . . . .	31
<b>3</b>	<b>API reference</b>	<b>33</b>
3.1	Client class . . . . .	33
3.2	Events and filters . . . . .	60
3.3	Types . . . . .	67
3.4	Sessions . . . . .	90
<b>4</b>	<b>Development resources</b>	<b>97</b>
4.1	Changelog (Version History) . . . . .	97
4.2	Migrating from v1 to v2 . . . . .	98
4.3	Frequently Asked Questions . . . . .	106
4.4	Contributing . . . . .	106
	<b>Python Module Index</b>	<b>111</b>
	<b>Index</b>	<b>113</b>



## Welcome to Telethon's documentation!

```
import asyncio
from telethon import Client, events
from telethon.events import filters

async def main():
    async with Client('name', api_id, api_hash) as client:
        me = await client.interactive_login()
        await client.send_message(me, f'Hello, {me.name}!')

        @client.on(events.NewMessage, filters.Text(r'(?i)hello'))
        async def handler(event):
            await event.reply('Hey!')

        await client.run_until_disconnected()

asyncio.run(main())
```

- Are you new here? Jump straight into [Installation!](#)
- Looking for the Client API reference? See the [Client class](#).
- Did you upgrade the library? Please read [Changelog \(Version History\)](#).
- Coming from Bot API or want to create new bots? See [HTTP Bot API vs MTProto](#).
- Used Telethon before v2.0? See [Migrating from v1 to v2](#).
- Want to hack away with the raw API? Search in [Telethon Raw API](#).

## What is this?

Telegram is a popular messaging application. This library is meant to make it easy for you to write Python programs that can interact with Telegram. Think of it as a wrapper that has already done the hard work for you, so you can focus on developing an application.

## How should I use the documentation?

This documentation is divided in multiple sections. The first few sections are a guide, while others contain the API reference or a glossary of terms. The documentation assumes some familiarity with Python.

If you are getting started with the library, you should follow the documentation in order by pressing the “Next” button at the bottom-right of every page.

You can also use the menu on the left to quickly skip over sections if you're looking for something in particular or want to continue where you left off.



## FIRST STEPS

In this section you will learn how to install the library and login to your Telegram account.

→ *Start reading Installation*

### 1.1 Installation

Telethon is a Python 3 library, which means you need to download and install Python to use it. Installing Python, using virtual environments, and the basics of the language, are outside of the scope of this guide.

You can find the official resources to [download Python](#), learn about the [Python Setup and Usage](#) for different platforms, or follow the [The Python Tutorial](#) to learn the basics. These are not necessarily the best resources to learn, but they are official. Be sure to search online if you prefer learning in video form or otherwise.

You can confirm that you have Python installed with:

```
python --version
```

Which should print something similar to `Python 3.11.5` (or newer). Be sure to run the command in a terminal such as PowerShell or Terminal. The above won't work inside a Python shell! If you had the terminal open before installing Python, you will probably need to open a new one.

#### 1.1.1 Installing the latest stable version

Once you have a working Python 3 installation, you can install or upgrade the telethon package with pip:

```
python -m pip install --upgrade "telethon~=2.0"
```

Be sure to use lock-files if your project! The above is just a quick way to get started and install a [v2-compatible](#) Telethon globally.

## 1.1.2 Installing development versions

If you want the *latest* unreleased changes, you can run the following command instead:

```
python -m pip install --upgrade https://github.com/LonamiWebs/Telethon/archive/v2.zip
↪#subdirectory=client
```

---

**Note:** The development version may have bugs and is not recommended for production use. However, when you are reporting a library bug, you must reproduce the issue in this version before reporting the problem.

---

## 1.1.3 Verifying the installation

To verify that the library is installed correctly, run the following command:

```
python -c "import telethon; print(telethon.__version__)"
```

The version number of the library should show in the output.

## 1.2 Signing in

Most of Telegram's API methods are gated behind an account login. But before you can interact with the API at all, you will need to obtain an API ID and hash pair for your application.

### 1.2.1 Registering your Telegram application

Before working with Telegram's API, you (as the application developer) need to get an API ID and hash:

1. [Login to your Telegram account](#) with the phone number of the developer account to use.
2. Click under *API Development tools*.
3. A *Create new application* window will appear. Fill in your application details. There is no need to enter any *URL*, and only the first two fields (*App title* and *Short name*) can currently be changed later.
4. Click on *Create application* at the end. Remember that your **API hash is secret** and Telegram won't let you revoke it. Don't post it anywhere!

This API ID and hash can now be used to develop an application using Telegram's API. Telethon consumes this API ID and hash in order to make the requests to Telegram.

It is important to note that this API ID and hash is attached to a **developer account**, and can be used to develop applications or otherwise using libraries such as Telethon.

The *users* of the application you develop do **not** need to provide their own API ID and hash. The API ID and hash values are meant to be hardcoded in the application. Any user is then able to login with just their phone number or bot token, even if they have not registered an application themselves.

---

**Important:** The API ID and hash are meant to be *secret*, but Python is often distributed in source-code form. These two things conflict with each other! You can opt to obfuscate the values somehow, or perhaps distribute an executable binary file instead. Depending on what you are developing, it might be reasonable to expect users to provide their own API ID and hash instead.



Official applications *also* must embed the API ID and hash, but these are often distributed as binary files. Whatever you do, **do not use other people's API ID and hash!** Telegram may detect this as suspicious and ban the accounts.

If you receive an error, Telegram is likely blocking the registration of a new applications. The best you can do is wait and try again later. If the issue persists, you may try contacting them, using a proxy or using a VPN. Be aware that some phone numbers are not eligible to register applications with.

## 1.2.2 Interactive login

The library offers a method for “quick and dirty” scripts known as `interactive_login()`. This method will first check whether the account was previously logged-in, and if not, ask for a phone number to be input.

You can write the code in a file (such as `hello.py`) and then run it, or use the built-in `asyncio`-enabled REPL. For this tutorial, we'll be using the `asyncio` REPL:

```
python -m asyncio
```

**Important:** If you opt to write your code in a file, do **not** call your script `telethon.py`! Python will try to import from there and it will fail with an error such as “`ImportError: cannot import name ...`”.

The first thing we need to do is import the `Client` class and create an instance of it:

```
from telethon import Client

client = Client('name', 12345, '0123456789abcdef0123456789abcdef')
```

The second and third parameters must be the API ID and hash, respectively. We have a client instance now, but we can't send requests to Telegram until we connect! So the next step is to `connect()`:

```
await client.connect()
```

If all went well, you will have connected to one of Telegram's servers. If you run into issues, you might need to try a different hosting provider or use some sort of proxy.

Once you're connected, we can begin the `interactive_login()`:

```
await client.interactive_login()
```

Do as the prompts say on the terminal, and you will have successfully logged-in!

Once you're done, make sure to `disconnect()` for a graceful shutdown.

To summarize:

```
from telethon import Client
client = Client('name', 12345, '0123456789abcdef0123456789abcdef')
await client.connect()
await client.interactive_login()
```

If you want to automatically login as a bot when needed, you can do so without any prompts, too:

```
from telethon import Client
client = Client('name', 12345, '0123456789abcdef0123456789abcdef')
```

(continues on next page)

(continued from previous page)

```
await client.connect()
await client.interactive_login('54321:hJrIQtVBab0M2Yqg4HL1K-EubfY_v2fEVR')
```

---

**Note:** The bot token obtained from [@BotFather](#) looks something like this:

```
54321:hJrIQtVBab0M2Yqg4HL1K-EubfY_v2fEVR
```

This is **not** the API ID and hash separated by a colon! All of it is the bot token. Using a bot with Telethon still requires a separate API ID and hash.

See [HTTP Bot API vs MTPProto](#) for more details.

---

### 1.2.3 Manual login

---

**Tip:** You can safely skip to *Next steps* if you’ve already completed the *Interactive login*. This section is only of interest if you want more control over how to manually login.

---

We’ve talked about the second and third parameters of the *Client* constructor, but not the first:

```
client = Client('name', 12345, '0123456789abcdef0123456789abcdef')
```

The first parameter is the “session”. When using a string or a *Path*, the library will create a SQLite database in that path. The session path can contain directory separators and live anywhere in the file system. Telethon will automatically append the `.session` extension if you don’t provide any.

Briefly, the session contains some of the information needed to connect to Telegram. This includes the data center belonging to the account logged-in, and the authorization key used for encryption, among other things.

---

**Important:** **Do not leak the session file!** Anyone with that file can login to the account stored in it. If you believe someone else has obtained this file, immediately revoke all active sessions from an official client.

---

Let’s take a look at what `interactive_login()` does under the hood.

1. First, it’s using an equivalent of `is_authorized()` to check whether the session was logged-in previously.
2. Then, it will either `bot_sign_in()` with a bot token or `request_login_code()` with a phone number.
  - If it logged-in as a bot account, a *User* is returned and we’re done.
  - Otherwise, a login code was sent. Go to step 3.
3. Attempt to complete the user sign-in with `sign_in()`, by entering the login code.
  - If a *User* is returned, we’re done.
  - Otherwise, a 2FA password is required. Go to step 4.
4. Use `Client.check_password()` to check that the password is correct.
  - If the password is correct, *User* is returned and we’re done.

Put into code, a user can thus login as follows:

```

from telethon import Client
from telethon.types import User

# SESSION, API_ID, API_HASH should be previously defined in your code
async with Client(SESSION, API_ID, API_HASH) as client:
    if not await client.is_authorized():
        phone = input('phone: ')
        login_token = await client.request_login_code(phone_or_token)

        code = input('code: ')
        user_or_token = await client.sign_in(login_token, code)

        if isinstance(user_or_token, User):
            return user_or_token

        # user_or_token is PasswordToken
        password_token = user_or_token

        import getpass
        password = getpass.getpass("password: ")
        user = await client.check_password(password_token, password)

... # can now use the client and user

```

A bot account does not need to request login code and cannot have passwords, so the login flow is much simpler:

```

from telethon import Client

# SESSION, API_ID, API_HASH should be previously defined in your code
async with Client(SESSION, API_ID, API_HASH) as client:
    if not await client.is_authorized():
        bot_token = input('token: ')
        bot_user = await client.bot_sign_in(bot_token)
        bot_user

... # can now use the client and bot_user

```

To get a bot account, you need to talk with @BotFather.

You may have noticed the `async with` keywords. The `Client` can be used in a context-manager. This will automatically call `Client.connect()` and `Client.disconnect()` for you.

A good way to structure your code is as follows:

```

import asyncio
from telethon import Client

SESSION = ...
API_ID = ...
API_HASH = ...

async def main():
    async with Client(SESSION, API_ID, API_HASH) as client:
        ... # use client to your heart's content

```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':  
    asyncio.run(main())
```

This way, both the `asyncio` event loop and the `Client` will exit cleanly. Otherwise, you might run into errors such as tasks being destroyed while pending.

---

**Note:** Once a `Client` instance has been connected, you cannot change the `asyncio` event loop. Methods like `asyncio.run()` setup and tear-down a new event loop every time. If the loop changes, the client is likely to be “stuck” because its loop cannot advance.

If you want to learn how `asyncio` works under the hood or need an introduction, you can read my own blog post [An Introduction to Asyncio](#).

---

## 1.3 Next steps

By now, you should have successfully gone through both the *Installation* and *Signing in* processes.

With a `Client` instance connected and authorized, you can send any request to Telegram. Some requests are bot-specific, and some are user-specific, but most can be used by any account. You will need to have the correct permissions and pass valid parameters, but after that, your imagination is the limit.

Telethon features extensive documentation for every public item offered by the library. All methods within the `Client` also contain one or more examples on how to use them.

Whatever you build, remember to comply with both [Telegram’s Terms of Service](#) and [Telegram’s API ToS](#). There are [several requests that applications must make](#):

[...] when logging in as an existing user, apps are supposed to call `help.getTermsOfServiceUpdate` to check for any updates to the Terms of Service; this call should be repeated after `expires` seconds have elapsed. If an update to the Terms Of Service is available, clients are supposed to show a consent popup; if accepted, clients should call `help.acceptTermsOfService`, providing the `termsOfService id` JSON object; in case of denial, clients are to delete the account using `account.deleteAccount`, providing Decline ToS update as deletion reason.

The library will not make these calls for you, as it cannot know how users interact with the application being developed. If you use an official client alongside the application you are developing, it should be safe to rely on that client making the requests instead.

Having your account banned might sound scary. However, keep in mind that people often don’t post comments when things work fine! The only comments you’re likely to see are negative ones. As long as you use a real phone number and don’t abuse the API, you will most likely be fine. This library would not be used at all otherwise!

If you believe your account was banned on accident, [there are ways to try to get it back](#).

If you are using a bot account instead, the risk of a ban is either zero or very close to it. If you know of a bot causing account bans, please let me know so it can be documented.

## CONCEPTS

A more in-depth explanation of some of the concepts and words used in Telethon.

→ *Start reading Chat concept*

### 2.1 Chats

The term *chat* is extremely overloaded, so it's no surprise many are confused by what it means. This section should hopefully clear that up.

#### 2.1.1 Telethon Chat

The word *chat* in Telethon is used to refer a place where messages are sent to. Therefore, a Telethon *chat* can be another user, a bot, a group, or a broadcast channel. All of those are places where messages can be sent.

Of course, chats do more things than contain messages. They often have a name, username, photo, description, and other information.

When a *chat* appears in a parameter or as a property, it means that it will be either a *User*, *Group* or *Channel*.

When a parameter must be “chat-like”, it means Telethon will accept anything that can be “converted” to a *chat*. The following types are chat-like:

- The 'me' literal string. This represents the account that is logged in (“yourself”).
- An '@username'. The at-sign @ is optional. Note that links are not supported.
- An '+1 23' phone number string. It must be an `str` and start with the plus-sign + character.
- An 123 integer identifier. It must be an `int` and cannot be negative.
- An existing *User*, *Group* or *Channel*.
- A *PackedChat*.

Previous versions of Telethon referred to this term as “entity” or “entities” instead.

## 2.1.2 Telegram Chat

The Telegram API is very confusing when it comes to the word “chat”. You only need to know about this if you plan to use the *Raw API*.

In the *TL* schema definitions, there are two boxed types, `User` and `Chat`. A boxed `User` can only be the bare `user`, but the boxed `Chat` can be either a bare `chat` or a bare `channel`.

A bare `chat` always refers to small groups. A bare `channel` can have either the `broadcast` or the `megagroup` flag set to `True`.

A bare `channel` with the `broadcast` flag set to `True` is known as a broadcast channel. A bare `channel` with the `megagroup` flag set to `True` is known as a supergroup.

A bare `chat` has less features available than a bare `channel megagroup`. Official clients are very good at hiding this difference. They will implicitly convert bare `chat` to bare `channel megagroup` when doing certain operations. Doing things like setting a username is actually a two-step process (migration followed by updating the username). Official clients transparently merge the history of migrated `channel` with their old `chat`.

In Telethon:

- A `User` always corresponds to `user`.
- A `Group` represents either a `chat` or a `channel megagroup`.
- A `Channel` represents a `channel broadcast`.

Telethon classes aim to map to similar concepts in official applications.

## 2.1.3 Bot API chat

The Bot API follows a certain convention when it comes to identifiers:

- User IDs are positive.
- Chat IDs are negative.
- Channel IDs are *also* negative, but are prefixed by `-100`.

Telethon encourages the use of `PackedChat` instead of naked identifiers. As a reminder, negative identifiers are not supported in Telethon’s chat-like parameters.

If you got an Bot API-style ID from somewhere else, you will need to explicitly say what type it is:

```
# If -1001234 is your ID...
from telethon.types import PackedChat, PackedType
chat = PackedChat(PackedType.BROADCAST, 1234, None)
# ...you need to explicitly create a PackedChat with id=1234 and set the corresponding
↪ type (a channel).
# The access hash (see below) will be ``None``, which may or may not work.
```

## 2.1.4 Encountering chats

The way you encounter chats in Telethon is no different from official clients. If you:

- ...have joined a group or channel, or have sent private messages to some user, you can `get_dialogs()`.
- ...know the user is in your contact list, you can `get_contacts()`.
- ...know the user has a common chat with you, you can `get_participants()` of the chat in common.
- ...know the username of the user, group, or channel, you can `resolve_username()`.
- ...are a bot responding to users, you will be able to access the `types.Message.sender`.

## 2.1.5 Chats access hash

Users, supergroups and channels all need an *access hash*.

In Telethon, the `PackedChat` is the recommended way to deal with the identifier-hash pairs. This compact type can be used anywhere a chat is expected. It's designed to be easy to store and cache in any way your application chooses.

Bot accounts can get away with an invalid *access hash* for certain operations under certain conditions. The same is true for user accounts, although to a lesser extent.

When using just the identifier to refer to a chat, Telethon will attempt to retrieve its hash from its in-memory cache. If this fails, an invalid hash will be used. This may or may not make the API call succeed. For this reason, it is recommended that you always use `PackedChat` instead.

Remember that an *access hash* is account-bound. You cannot obtain an *access hash* in Account-A and use it in Account-B.

## 2.2 Updates

Updates are an important topic in a messaging platform like Telegram. After all, you want to be notified as soon as certain events happen, such as new message arrives.

Telethon abstracts away Telegram updates with *events*.

---

**Important:** It is strongly advised to configure logging when working with events:

```
import logging
logging.basicConfig(
    format='[%(levelname) 5s/%(asctime)s] %(name)s: %(message)s',
    level=logging.WARNING
)
```

With the above, you will see all warnings and errors and when they happened.

---

## 2.2.1 Listening to updates

You can define and register your own functions to be called when certain `telethon.events` occur.

The most common way is using the `Client.on()` decorator to register your callback functions, often referred to as *handlers*:

```
from telethon import Client, events
from telethon.events import filters

bot = Client(...)

@bot.on(events.NewMessage, filters.Command('/start') | filters.Command('/help'))
async def handler(event: events.NewMessage):
    await event.respond('Beep boop!')
```

The first parameter is the *type* of one of the `telethon.events`, not an instance, so make sure you don't write parenthesis after it.

The second parameter is optional. If provided, it must be a callable function that returns `True` if the handler should run. Built-in filter functions are available in the `filters` module. In this example, `Command` means the handler will be called when the user sends `/start` or `/help` to the bot.

Built-in filter functions are also *Combinable*. This means you can use `|`, `&` and the unary `~` to combine filters with *or*, *and*, and negate them, respectively. These operators correspond to `events.filters.Any`, `events.filters.All` and `events.filters.Not`.

When your handler function is called, it will receive a single parameter, the event. The event type is the same as the one you defined in the decorator when registering your handler. You don't need to explicitly set the type hint, but you can do so if you want your IDE to assist in autocompletion.

If you cannot use decorators, you can use the `Client.add_event_handler()` method instead. The above code is equivalent to the following:

```
from telethon import Client, events
from telethon.events import filters

async def handler(event: events.NewMessage):
    await event.respond('Beep boop!')

bot = Client(...)
bot.add_event_handler(handler, events.NewMessage, filters.Command('/start'))
```

Note how the above lets you defined the `Client` instance *after* your handlers. In other words, you can define your handlers without the `Client` instance. This may make it easier to place them in a separate file.

## 2.2.2 Filtering events

There is no way to tell Telegram to only send certain updates. Telegram sends all updates to connected active clients as they occur. Telethon must be received and process all updates to ensure correct ordering.

Filters are not magic. They work all the same as `if` conditions inside your event handlers. However, they offer a more convenient and consistent way to check for certain conditions.

All built-in filters can be found in `telethon.events.filters`.

When registering an event handler, you can optionally define the filter to use. You can retrieve a handler's filter with `get_handler_filter()`. You can set (and overwrite) a handler's filter with `set_handler_filter()`.



Filters are meant to be fast and never raise exceptions. For this reason, filters cannot be asynchronous. This reduces the chance a filter will do slow IO and potentially fail.

A filter is simply a callable function that takes an event as input and returns a boolean. If the filter returns `True`, the handler will be called. Using this knowledge, you can create custom filters too. If you need state, you can use a class with a `__call__` method defined:

```
# Anonymous filter which only handles messages with ID = 1000
bot.add_event_handler(handler, events.NewMessage, lambda e: e.id == 1000)
#           this parameter is the filter ^-----^

# ...

def only_odd_messages(event):
    "A filter that only handles messages when their ID is divisible by 2"
    return event.id % 2 == 0

bot.add_event_handler(handler, events.NewMessage, only_odd_messages)

# ...

class OnlyDivisibleMessages:
    "A filter that only handles messages when their ID is divisible by some amount"
    def __init__(self, divisible_by):
        self.divisible_by = divisible_by

    def __call__(self, event):
        return event.id % self.divisible_by == 0

bot.add_event_handler(handler, events.NewMessage, OnlyDivisibleMessages(7))
```

Custom filters should accept any `Event`. You can use `isinstance()` if your filter can only deal with certain types of events.

If you need to perform asynchronous operations, you can't use a filter. Instead, manually check for those conditions inside your handler.

The filters work all the same when using `Client.on()`. This makes it very convenient to write custom filters using the `lambda` syntax:

```
@bot.on(events.NewMessage, lambda e: e.id == 1000)
async def handler(event):
    ...
```

### 2.2.3 Setting priority on handlers

There is no explicit way to define a different priority for different handlers.

Instead, the library will call all your handlers in the order you added them. This means that, if you want a “catch-all” handler, it should be registered last.

By default, the library will stop calling the rest of handlers after one is called:

```
@bot.on(events.NewMessage)
async def first(event):
```

(continues on next page)

(continued from previous page)

```

    print('This is always called on new messages!')

@bot.on(events.NewMessage)
async def second(event):
    print('This will never be called, because "first" already ran.')

```

This is often the desired behaviour if you're using filters.

If you have more complicated filters executed *inside* the handler, Telethon believes your handler completed and will stop calling the rest. If that's the case, you can `return events.Continue`:

```

@bot.on(events.NewMessage)
async def first(event):
    print('This is always called on new messages!')
    return events.Continue

@bot.on(events.NewMessage)
async def second(event):
    print('Now this one runs as well!')

```

Alternatively, if this is *always* the behaviour you want, you can configure it in the `Client`:

```

bot = Client(..., check_all_handlers=True)
#
# Now the code above will call both handlers, even without returning events.Continue

```

If you need a more complicated setup, consider sorting all your handlers beforehand. Then, use `Client.add_event_handler()` on all of them to ensure the correct order.

## 2.3 Messages

Messages are at the heart of a messaging platform. In Telethon, you will be using the `Message` class to interact with them.

### 2.3.1 Fetching messages

The most common way to actively fetch messages using the `Client.get_messages()` method:

```

# Get the last message in a chat (by setting the limit to 1).
last_message = (await client.get_messages(chat, 1))[0]

# Iterate over all messages in a chat, starting from the oldest message (by using
↪reversed).
async for message in reversed(client.get_messages(chat)):
    print(message.sender.name, message.text_html)

```

You can also perform a fuzzy text search with the `Client.search_messages()` method. The search will be performed server-side by Telegram, so the rules for how it works are also fuzzy.

If you want to search for messages in all the chats you're part of, you can use `Client.search_all_messages()`.

Lastly, `Client.send_message()` also returns the `Message` that you just sent.

The most common way to passively listen to incoming messages is using the `NewMessage` event:

```
from telethon import events

@client.on(events.NewMessage)
async def first(event):
    print(event.chat.name, ':', event.text)
```

**See also:**

The `Updates` concept for an in-depth explanation on using events.

### 2.3.2 Formatting messages

The library supports 3 formatting modes: no formatting, CommonMark, HTML.

Telegram does not natively support markdown or HTML. Clients such as Telethon parse the text into a list of formatting `MessageEntity` at different offsets.

Note that CommonMark's markdown is not fully compatible with `HTTP Bot API's MarkdownV2` style, and does not support spoilers:

```
*italic* and _italic_
**bold** and __bold__
# headings are underlined
~~strikethrough~~
[inline URL](https://www.example.com/)
[inline mention](tg://user?id=ab1234cd6789)
custom emoji image with 
`inline code`
```python
multiline pre-formatted
block with optional language
```
```

HTML is also not fully compatible with `HTTP Bot API's HTML` style, and instead favours more standard HTML elements:

- `strong` and `b` for **bold**.
- `em` and `i` for *italics*.
- `u` for underlined text.
- `del` and `s` for strikethrough.
- `blockquote` for quotes.
- `details` for hidden text (spoiler).
- `code` for inline code
- `pre` for multiple lines of code.
- `a` for links.
- `img` for inline images (only custom emoji).

Both markdown and HTML recognise the following special URLs using the `tg:` protocol:

- `tg://user?id=ab1234cd6789` for inline mentions. To make sure the mention works, use `types.PackedChat.hex`. You can also use `types.User.id`, but the mention will fail if the user is not in cache.
- `tg://emoji?id=1234567890` for custom emoji. You must use the document identifier as the value. The alt-text of the image **must** be a emoji such as `.`

To obtain a message's text formatted, use `types.Message.text_markdown` or `types.Message.text_html`.

To send a message with formatted text, use the `markdown` or `html` parameters in `Client.send_message()`.

When sending files, the format is appended to the name of the `caption` parameter, either `caption_markdown` or `caption_html`.

### Link previews

Link previews are treated as a type of media automatically generated by Telegram. This means you cannot have both a link preview and other media in the same message.

The `link_preview` parameter indicates whether link previews are *allowed* to be present. If Telegram is unable to generate a link preview for any of the links, there won't be a link preview.

By default, link previews are not enabled. This is done to prevent sending things you did not explicitly intend to send. Unlike the official clients, which do not have a GUI to “enable” the preview, you can easily enable them from code.

Telegram will attempt to generate a preview for all links contained in the message in order. You can use this to your advantage, and hide a link to a photo in the first space or invisible character like `'\u2063'`. Note that avid users *will* be able to find out the link. It is not secret!

Link previews of photos won't show under the photos of the chat, but it requires a server hosting the image, a public address, and Telegram to be able to generate a preview.

To regenerate a preview, send the corresponding link to `@WebpageBot`.

### 2.3.3 Message identifiers

This is an in-depth explanation for how the `types.Message.id` works.

---

**Note:** You can safely skip this section if you're not interested.

---

Every account, whether it's an user account or bot account, has its own message counter. This counter starts at 1, and is incremented by 1 every time a new message is received. In private conversations or small groups, each account will receive a copy each message. The message identifier will be based on the message counter of the receiving account.

In megagroups and broadcast channels, the message counter instead belongs to the channel itself. It also starts at 1 and is incremented by 1 for every message sent to the group or channel. This means every account will see the same message identifier for a given message in a group or channel.

This design has the following implications:

- The message identifier alone is enough to uniquely identify a message only if it's not from a megagroup or channel. This is why `events.MessageDeleted` does not need to (and doesn't always) include chat information.
- Messages cannot be deleted for one-side only in megagroups or channels. Because every account shares the same identifier for the message, it cannot be deleted only for some.
- Links to messages only work for everyone inside megagroups or channels. In private conversations and small groups, each account will have their own counter, and the identifiers won't match.

Let's look at a concrete example.

- You are logged in as User-A.
- Both User-B and User-C are your mutual contacts.
- You have share a small group called Group-S with User-B.
- You also share a megagroup called Group-M with User-C.

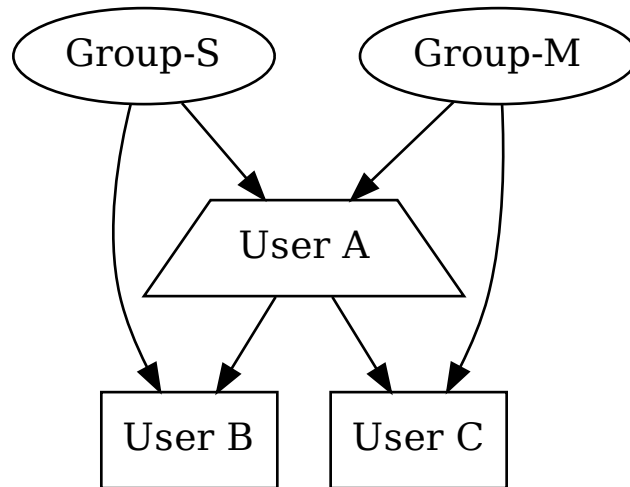


Fig. 1: Demo scenario

Every account and channel has just been created. This means everyone has a message counter of one.

First, User-A will sent a welcome message to both User-B and User-C:

```
User-A → User-B: Hey, welcome!
User-A → User-C: ¡Bienvenido!
```

- For User-A, “Hey, welcome!” will have the message identifier 1. The message with “¡Bienvenido!” will have an ID of 2.
- For User-B, “Hey, welcome” will have ID 1.
- For User-B, “¡Bienvenido!” will have ID 1.

Table 1: Message identifiers

| Message       | User-A | User-B | User-C | Group-S | Group-M |
|---------------|--------|--------|--------|---------|---------|
| Hey, welcome! | 1      | 1      |        |         |         |
| ¡Bienvenido!  | 2      |        | 1      |         |         |

Next, User-B and User-C will respond to User-A:

```
User-B → User-A: Thanks!
User-C → User-A: Gracias :)
```

Table 2: Message identifiers

| Message       | User-A | User-B | User-C | Group-S | Group-M |
|---------------|--------|--------|--------|---------|---------|
| Hey, welcome! | 1      | 1      |        |         |         |
| ¡Bienvenido!  | 2      |        | 1      |         |         |
| Thanks!       | 3      | 2      |        |         |         |
| Gracias :)    | 4      |        | 2      |         |         |

Notice how for each message, the counter goes up by one, and they are independent.

Let's see what happens when User-B sends a message to Group-S:

User-B → Group-S: Nice group

Table 3: Message identifiers

| Message       | User-A | User-B | User-C | Group-S | Group-M |
|---------------|--------|--------|--------|---------|---------|
| Hey, welcome! | 1      | 1      |        |         |         |
| ¡Bienvenido!  | 2      |        | 1      |         |         |
| Thanks!       | 3      | 2      |        |         |         |
| Gracias :)    | 4      |        | 2      |         |         |
| Nice group    | 5      | 3      |        |         |         |

While the message was sent to a different chat, the group itself doesn't have a counter. The message identifiers are still unique for each account. The chat where the message was sent can be completely ignored.

Megagroups behave differently:

User-C → Group-M: Buen grupo

Table 4: Message identifiers

| Message       | User-A | User-B | User-C | Group-S | Group-M |
|---------------|--------|--------|--------|---------|---------|
| Hey, welcome! | 1      | 1      |        |         |         |
| ¡Bienvenido!  | 2      |        | 1      |         |         |
| Thanks!       | 3      | 2      |        |         |         |
| Gracias :)    | 4      |        | 2      |         |         |
| Nice group    | 5      | 3      |        |         |         |
| Buen grupo    |        |        |        |         | 1       |

The group has its own message counter. Each user won't get a copy of the message with their own identifier, but rather everyone sees the same message.

## 2.4 Sessions

In Telethon, the word *session* is used to refer to the set of data needed to connect to Telegram. This includes the server address of your home data center, as well as the authorization key bound to an account. When you first connect to Telegram, an authorization key is generated to encrypt all communication. After login, Telegram remembers this authorization key as logged-in, so you don't need to login again.

---

**Important: Do not leak the session file!** Anyone with that file can login to the account stored in it. If you believe someone else has obtained this file, immediately revoke all active sessions from an official client.

---

Some auxiliary information such as the user ID of the logged-in user is also kept.

The update state, which can change every time an update is received from Telegram, is also stored in the session. Telethon needs this information to catch up on all missed updates while your code was not running. This is why it's important to call `Client.disconnect()`. Doing so flushes all the update state to the session and saves it.

### 2.4.1 Session files

Telethon defaults to using SQLite to store the session state. The session state is written to `.session` files, so make sure your VCS ignores them! To make sure the `.session` file is saved, you should call `Client.disconnect()` before exiting the program.

The first parameter in the `Client` constructor is the session to use. You can use a `str`, a `pathlib.Path` or a `session.Storage`. The string or path are relative to the Current Working Directory. You can use absolute paths or relative paths to folders elsewhere. The `.session` extension is automatically added if the path has no extension.

### 2.4.2 Session storages

The `session.Storage` abstract base class defines the required methods to create custom storages. Telethon comes with two built-in storages:

- `SqliteSession`. This is used by default when a string or path is used.
- `MemorySession`. This is used by default when the path is `None`. You can also use it directly when you have a `Session` instance. It's useful when you don't have file-system access.

If you would like to store the session state in a different way, you can subclass `session.Storage`. You may also find [custom third-party session storages in Telethon's wiki](#). Be careful with any third-party code you install, as they could steal the login credentials. Only use session storages you trust, and pin the specific versions you have audited.

Some Python installations do not have the `sqlite3` module. In this case, attempting to use the default `SqliteSession` will fail. If this happens, you can try reinstalling Python. If you still don't have the `sqlite3` module, you should use a different storage.

## 2.5 RPC Errors

*RPC* stands for Remote Procedure Call. By extension, RPC Errors occur when a RPC fails to execute in the server. In Telethon, a *RPC error* corresponds to the *RpcError* class.

Telethon will only ever raise *RpcError* when the result to a *RPC* is an error. If the error is raised, you know it comes from Telegram. Consequently, when using *Raw API* directly, if a *RpcError* occurs, it is *extremely unlikely* to be a bug in the library. When *RpcErrors* are raised using the *Raw API*, Telegram is the one that decided an error should occur.

*RPC error* consist of an integer *code* and a string *name*. The *RpcError.code* is roughly the same as *HTTP status codes*. The *RpcError.name* is often a string in *SCREAMING\_CASE* and refers to what went wrong.

Certain error names also contain an integer value. This value is removed from the *name* and put into *RpcError.value*. If Telegram responds with *FLOOD\_WAIT\_60*, the name would be 'FLOOD\_WAIT' and the value 60.

A very common error is *FLOOD\_WAIT*. It occurs when you have attempted to use a request too many times during a certain window of time:

```
import asyncio
from telethon import errors

try:
    await client.send_message('me', 'Spam')
except errors.FloodWait as e:
    # A flood error; sleep.
    await asyncio.sleep(e.value)
```

Note that the library can automatically handle and retry on *FLOOD\_WAIT* for you. Refer to the *flood\_sleep\_threshold* of the *Client* to learn how.

Refer to the documentation of the *telethon.errors* pseudo-module for more details.

## 2.6 HTTP Bot API vs MTProto

Telethon is more than capable to develop bots for Telegram. If you haven't decided which wrapper library for bots to use yet, using Telethon from the beginning may save you some headaches later.

### 2.6.1 What is Bot API?

Telegram's *HTTP Bot API*, from now on referred to as simply "Bot API", is Telegram's official way for developers to control their own Telegram bots. Quoting their main page:

The *Bot API* is an HTTP-based interface created for developers keen on building bots for Telegram.

To learn how to create and set up a bot, please consult our [Introduction to Bots](#) and [Bot FAQ](#).

*Bot API* is simply an HTTP endpoint offering a custom HTTP API. Underneath, it uses *tdlib* to talk to Telegram's servers.

You can configure your bot details via [@BotFather](#). This includes name, commands, and auto-completion.



## 2.6.2 What is MTPROTO?

**MTPROTO** stands for “Mobile Transport Protocol”. It is the language that the Telegram servers “speak”. You can think of it as an alternative to HTTP.

Telegram offers multiple APIs. All user accounts must use the API offered via MTPROTO. We will call this API the “MTPROTO API”. This is the canonical Telegram API.

The MTPROTO API is different from Bot API, but bot accounts can use either in the same way. In fact, the Bot API is implemented to use the MTPROTO API to map the requests and responses.

Telethon implements the MTPROTO and offers classes and methods that can be called to send requests. In Telethon, all the methods and types generated from Telegram’s API definitions are also known as *Raw API*. This name was chosen because it gives you “raw” access to the MTPROTO API. Telethon’s *Client* and other custom types are implemented using the *Raw API*.

## 2.6.3 Why is an API ID and hash needed for bots with MTPROTO?

When talking to Telegram’s API directly, you need an API ID and hash to sign in to their servers. API access is forbidden without an API ID, and the sign in can only be done with the API hash.

When using the *Bot API*, that layer talks to the MTPROTO API underneath. To do so, it uses its own private API ID and hash.

When you cut on the intermediary, you need to provide your own. In a similar manner, the authorization key which remembers that you logged-in must be kept locally.

## 2.6.4 Advantages of MTPROTO over Bot API

MTPROTO clients (like Telethon) connect directly to Telegram’s servers via TCP or UDP. There is no HTTP connection, no “polling”, and no “web hooks”. We can compare the two visually:

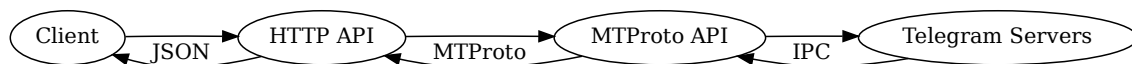


Fig. 2: Communication between a Client and the HTTP Bot API



Fig. 3: Communication between a Client and Telegram’s API via MTPROTO

When interacting with the MTPROTO API directly, we can cut down one intermediary (the HTTP API). This is less theoretical overhead and latency. It also means that, even if the Bot API endpoint is down, talking to the MTPROTO API could still work.

The methods offered by the Bot API map to some of the methods in the MTProto API, but not all. The Bot API is its own abstraction, and chooses to expose less details. By talking to the MTProto API directly, you unlock the [full potential](#).

The serialization format used by MTProto is more compact than JSON and can still be compressed.

Another benefit of avoiding the Bot API is the ease to switch to user accounts instead of bots. The MTProto API is the same for users and bots, so by using Telethon, you don't need to learn to use a second library.

### 2.6.5 Migrating from Bot API to Telethon

If the above points convinced you to switch to Telethon, the following short guides should help you make the switch!

It doesn't matter if you wrote your bot with [requests](#) and you were making API requests manually, or if you used a wrapper library like [python-telegram-bot](#) or [pyTelegramBotAPI](#). You will surely be pleased with Telethon!

If you were using an asynchronous library like [aiohttp](#) or a wrapper like [aiogram](#), the switch will be even easier.

#### Migrating from PTB v13.x

Using one of the examples from their v13 wiki with the `.ext` module:

```
from telegram import Update
from telegram.ext import Updater, CallbackContext, CommandHandler

updater = Updater(token='TOKEN', use_context=True)
dispatcher = updater.dispatcher

def start(update: Update, context: CallbackContext):
    context.bot.send_message(chat_id=update.effective_chat.id, text="I'm a bot, please.
↪talk to me!")

start_handler = CommandHandler('start', start)
dispatcher.add_handler(start_handler)

updater.start_polling()
```

The code creates an `Updater` instance. This will take care of polling updates for the bot associated with the given token. Then, a `CommandHandler` using our `start` function is added to the dispatcher. At the end, we block, telling the updater to do its job.

In Telethon:

```
import asyncio
from telethon import Client
from telethon.events import NewMessage, filters

updater = Client('bot', api_id, api_hash)

async def start(update: NewMessage):
    await update.client.send_message(chat=update.chat.id, text="I'm a bot, please talk.
↪to me!")

start_filter = filters.Command('/start')
```

(continues on next page)

(continued from previous page)

```

updater.add_event_handler(start, NewMessage, start_filter)

async def main():
    async with updater:
        await updater.interactive_login('TOKEN')
        await updater.run_until_disconnected()

asyncio.run(main())

```

Key differences:

- Telethon only has a *Client*, not separate Bot or Updater classes.
- There is no separate dispatcher. The *Client* is capable of dispatching updates.
- Telethon handlers only have one parameter, the event.
- There is no context, but the *client* property exists in all events.
- Handler types are *filters* and don't have a Handler suffix.
- Telethon must define the update type (*NewMessage*) and filter.
- The setup to run the client (and dispatch updates) is a bit more involved with *asyncio*.

Here's the above code in idiomatic Telethon:

```

import asyncio
from telethon import Client, events
from telethon.events import filters

client = Client('bot', api_id, api_hash)

@client.on(events.NewMessage, filters.Command('/start'))
async def start(event):
    await event.respond("I'm a bot, please talk to me!")

async def main():
    async with client:
        await client.interactive_login('TOKEN')
        await client.run_until_disconnected()

asyncio.run(main())

```

Events can be added using decorators and methods such as *types.Message.respond()* help reduce the verbosity.

## Migrating from PTB v20.x

Using one of the examples from their v13 wiki with the *.ext* module:

```

from telegram import Update
from telegram.ext import ApplicationBuilder, ContextTypes, CommandHandler

async def start(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await context.bot.send_message(chat_id=update.effective_chat.id, text="I'm a bot, ↵
↵please talk to me!")

```

(continues on next page)

(continued from previous page)

```

if __name__ == '__main__':
    application = ApplicationBuilder().token('TOKEN').build()

    start_handler = CommandHandler('start', start)
    application.add_handler(start_handler)

    application.run_polling()

```

No need to import the `asyncio` module directly! Now instead there are builders to help set stuff up.

In Telethon:

```

import asyncio
from telethon import Client
from telethon.events import NewMessage, filters

async def start(update: NewMessage):
    await update.client.send_message(chat=update.chat.id, text="I'm a bot, please talk_
↳to me!")

async def main():
    application = Client('bot', api_id, api_hash)

    start_filter = filters.Command('/start')
    application.add_event_handler(start, NewMessage, start_filter)

    async with application:
        await application.interactive_login('TOKEN')
        await application.run_until_disconnected()

asyncio.run(main())

```

Key differences:

- No builders. Telethon tries to get out of your way on how you structure your code.
- The client must be connected before it can run, hence the `async with`.

Here's the above code in idiomatic Telethon:

```

import asyncio
from telethon import Client, events
from telethon.events import filters

@client.on(events.NewMessage, filters.Command('/start'))
async def start(event):
    await event.respond("I'm a bot, please talk to me!")

async def main():
    async with Client('bot', api_id, api_hash) as client:
        await client.interactive_login('TOKEN')
        client.add_event_handler(start, NewMessage, filters.Command('/start'))
        await client.run_until_disconnected()

```

(continues on next page)

(continued from previous page)

```
asyncio.run(main())
```

Note how the client can be created and started in the same line. This makes it easy to have clean disconnections once the script exits.

## Migrating from asynchronous TeleBot

Using one of the examples from their v4 pyTelegramBotAPI documentation:

```
from telebot.async_telebot import AsyncTeleBot
bot = AsyncTeleBot('TOKEN')

# Handle '/start' and '/help'
@bot.message_handler(commands=['help', 'start'])
async def send_welcome(message):
    await bot.reply_to(message, """\
Hi there, I am EchoBot.
I am here to echo your kind words back to you. Just say anything nice and I'll say the
↳exact same thing to you!\
""")

# Handle all other messages with content_type 'text' (content_types defaults to ['text'])
@bot.message_handler(func=lambda message: True)
async def echo_message(message):
    await bot.reply_to(message, message.text)

import asyncio
asyncio.run(bot.polling())
```

This showcases a command handler and a catch-all echo handler, both added with decorators.

In Telethon:

```
from telethon import Client, events
from telethon.events.filters import Any, Command, Media
bot = Client('bot', api_id, api_hash)

# Handle '/start' and '/help'
@bot.on(events.NewMessage, Any(Command('/help'), Command('/start'))))
async def send_welcome(message: NewMessage):
    await message.reply("""\
Hi there, I am EchoBot.
I am here to echo your kind words back to you. Just say anything nice and I'll say the
↳exact same thing to you!\
""")

# Handle all other messages without media (negating the filter using ~)
@bot.on(events.NewMessage, ~Media())
async def echo_message(message: NewMessage):
    await message.reply(message.text)
```

(continues on next page)

(continued from previous page)

```
import asyncio
async def main():
    async with bot:
        await bot.interactive_login('TOKEN')
        await bot.run_until_disconnected()
asyncio.run(main())
```

Key differences:

- The handler type is defined using the event type instead of being a specific method in the client.
- Filters are also separate instances instead of being tied to specific event types.
- The `reply_to` helper is in the message, not the client instance.
- Setup is a bit more involved because the connection is not implicit.

For the most part, it's a 1-to-1 translation and the result is idiomatic Telethon.

## Migrating from aiogram

Using one of the examples from their v3 documentation with logging and comments removed:

```
import asyncio

from aiogram import Bot, Dispatcher, types
from aiogram.enums import ParseMode
from aiogram.filters import CommandStart
from aiogram.types import Message
from aiogram.utils.markdown import hbold

dp = Dispatcher()

@dp.message(CommandStart())
async def command_start_handler(message: Message) -> None:
    await message.answer(f"Hello, {hbold(message.from_user.full_name)}!")

@dp.message()
async def echo_handler(message: types.Message) -> None:
    try:
        await message.send_copy(chat_id=message.chat.id)
    except TypeError:
        await message.answer("Nice try!")

async def main() -> None:
    bot = Bot(TOKEN, parse_mode=ParseMode.HTML)
    await dp.start_polling(bot)

if __name__ == "__main__":
    asyncio.run(main())
```

We can see a specific handler for the `/start` command and a catch-all echo handler:

In Telethon:

```

import asyncio, html

from telethon import Client, RpcError, types, events
from telethon.events.filters import Command
from telethon.types import Message

client = Client("bot", api_id, api_hash)

@client.on(events.NewMessage, Command("/start"))
async def command_start_handler(message: Message) -> None:
    await message.respond(html=f"Hello, <b>{html.escape(message.sender.name)}</b>!")

@dp.message()
async def echo_handler(message: types.Message) -> None:
    try:
        await message.respond(message)
    except RpcError:
        await message.respond("Nice try!")

async def main() -> None:
    async with bot:
        await bot.interactive_login(TOKEN)
        await bot.run_until_disconnected()

if __name__ == "__main__":
    asyncio.run(main())

```

Key differences:

- There is no separate dispatcher. Handlers are added to the client.
- There is no specific handler for the `/start` command.
- The answer method is for callback queries. Messages have `respond()`.
- Telethon doesn't have functions to format messages. Instead, markdown or HTML are used.
- Telethon cannot have a default parse mode. Instead, it should be specified when responding.
- Telethon doesn't have `send_copy`. Instead, `Client.send_message()` accepts `Message`.
- If sending a message fails, the error will be `RpcError`, because it comes from Telegram.

## 2.7 The Full API

The API surface offered by Telethon is not exhaustive. Telegram is constantly adding new features, and both implementing and documenting custom methods would be an exhausting, never-ending job.

Telethon concedes to this fact and implements only commonly-used features to keep a lean API. Access to the entirety of Telegram's API via Telethon's *Raw API* is a necessary evil.

The `telethon._tl` module has a leading underscore to signal that it is private. It is not covered by the `semver` guarantees of the library, but you may need to use it regardless. If the `Client` doesn't offer a method for what you need, using the *Raw API* is inevitable.

## 2.7.1 Invoking Raw API methods

The *Raw API* can be *invoked* in a very similar way to other client methods:

```
from telethon import _tl as tl

was_reset = await client(tl.functions.account.reset_wall_papers())
```

Inside `telethon._tl.functions` you will find a function for every single *RPC* supported by Telegram. The parameters are keyword-only and do not have defaults. Whatever arguments you pass is exactly what Telegram will receive. Whatever is returned is exactly what Telegram responded with.

All functions inside `telethon._tl.functions` will return the serialized request. When calling a *Client* instance with this request as input, it will be sent to Telegram and wait for a response.

Multiple requests may be in-flight at the same time, specially when using `asyncio`. Telethon will attempt to combine these into a single “container” when possible as an optimization.

## 2.7.2 Exploring the Raw API

Everything under `telethon._tl.types` implements `repr()`. This means you can print any response and get the Python representation of that object.

All types are proper classes with attributes. You do not need to use a regular expression on the string representation to access the field you want.

Most *RPC* return an abstract class from `telethon._tl.abcs`. To check for a concrete type, you can use `isinstance()`:

```
invite = await client(tl.functions.messages.check_chat_invite(hash='aBcDeF'))
if isinstance(invite, tl.types.ChatInviteAlready):
    print(invite.chat)
```

The `telethon._tl` module is not documented here because it would greatly bloat the documentation and make search harder. Instead, there are multiple alternatives:

- Use Telethon’s separate site to search in the [Telethon Raw API](#). This is the recommended way. It also features auto-generated examples.
- Use Python’s built-in `help()` and `dir()` to help you navigate the module.
- Use an editor with autocompletion support.
- Choose the right layer from [Telegram’s official API Layers](#). Note that the [TL Schema](#) might not be up-to-date.

## 2.8 Data centers

Telegram has multiple servers, known as *data centers* or *MTPROTO* servers, all over the globe. This makes it possible to have reasonably low latency when sending messages.

When an account is created, Telegram chooses the most appropriated data center for you. This means you *cannot* change what your “home data center” is. However, [Telegram may change it after prolonged use from other locations](#).



## 2.8.1 Connecting behind a proxy

You can change the way Telethon opens a connection to Telegram's data center by setting a different *Connector*.

A connector is a function returning an asynchronous reader-writer pair. The default connector is `asyncio.open_connection()`, defined as:

```
def default_connector(ip: str, port: int):
    return asyncio.open_connection(ip, port)
```

While proxies are not directly supported in Telethon, you can change the connector to use a proxy. Any proxy library that supports `asyncio`, such as `python-socks[asyncio]`, can be used:

```
import asyncio
from functools import partial
from python_socks.async_.asyncio import Proxy
from telethon import Client

async def my_proxy_connector(ip, port, *, proxy_url):
    # Refer to python-socks for an up-to-date way to define and use proxies.
    # This is just an example of a custom connector.
    proxy = Proxy.from_url(proxy_url)
    sock = await proxy.connect(dest_host='example.com', dest_port=443)
    return await asyncio.open_connection(
        host=ip,
        port=port,
        sock=sock,
        ssl=ssl.create_default_context(),
        server_hostname='example.com',
    )

client = Client(..., connector=partial(
    my_proxy_connector,
    proxy_url='socks5://user:password@127.0.0.1:1080'
))
```

---

**Important:** Proxies can be used with Telethon, but they are not directly supported. Any connection errors you encounter while using a proxy are therefore very unlikely to be errors in Telethon. Connection errors when using custom connectors will *not* be considered bugs in the Telethon.

---

**Note:** Some proxies only support HTTP traffic. Telethon by default does not transmit HTTP-encoded packets. This means some HTTP-only proxies may not work.

---

## 2.8.2 Test servers

While you cannot change the production data center assigned to your account, you can tell Telethon to connect to a different server.

This is most useful to connect to the official Telegram test servers or *even your own*.

You need to import and define the `session.DataCenter` to connect to when creating the `Client`:

```
from telethon import Client
from telethon.session import DataCenter

client = Client(..., datacenter=DataCenter(id=2, ipv4_addr='149.154.167.40:443'))
```

This will override the value coming from the `Session`. You can get the test address for your account from [My Telegram](#).

**Note:** Make sure the `Sessions` you use for this client had not been created for the production servers before. The library will attempt to use the existing authorization key saved based on the data center identifier. This will most likely fail if you mix production and test servers.

There are public phone numbers anyone can use, with the following format:

Listing 1: 99966XYYYY test phone number, X being the datacenter identifier and YYYY random digits

```
99966 X YYYY
\___/ \_/ \___/
|   |   ` - random number
|   ` - datacenter identifier
` - fixed digits
```

For example, the test phone number 1234 for the datacenter 2 would be 9996621234.

The confirmation code to complete the login is the datacenter identifier repeated five times, in this case, 22222.

Therefore, it is possible to automate the login procedure, assuming the account exists and there is no 2-factor authentication:

```
from random import randrange
from telethon import Client
from telethon.session import DataCenter

datacenter = DataCenter(id=2, ipv4_addr='149.154.167.40:443')
phone = f'{randrange(1, 9999):04}'
login_code = str(datacenter.id) * 5
client = Client(..., datacenter=datacenter)

async with client:
    if not await client.is_authorized():
        login_token = await client.request_login_code(phone_or_token)
        await client.sign_in(login_token, login_code)
```

## 2.9 Glossary

### access hash

Account-bound integer tied to a specific resource. Users, channels, photos and documents are all resources with an access hash. The access hash doesn't change, but every account will see a different value for the same resource.

### Bot API

#### HTTP Bot API

Telegram's simplified HTTP API to control bot accounts only.

#### See also:

The *HTTP Bot API vs MTPProto* concept.

### chat

A *User*, *Group* or *Channel*.

#### See also:

The *Chats* concept.

### layer

When Telegram releases new features, it does so by releasing a new "layer". The different layers let Telegram know what a client is capable of and how it should respond to requests.

### login

Used to refer to the login process as a whole, as opposed to the action to *sign in*. The "login code" or "login token" get their name because they belong to the login process.

### MTPProto

Mobile Transport Protocol used to interact with Telegram's API.

#### See also:

The *HTTP Bot API vs MTPProto* concept.

### Raw API

Functions and types under `telethon._tl` that enable access to all of Telegram's API.

#### See also:

The *The Full API* concept.

### RPC

Remote Procedure Call. Invoked when calling a *Client* with a function from `telethon._tl.functions`.

### RPC Error

Error type returned by Telegram. *RpcError* contains an integer code similar to HTTP status codes and a name.

#### See also:

The *RPC Errors* concept.

### session

Data used to securely connect to Telegram and other state related to the logged-in account.

#### See also:

The *Sessions* concept.

### sign in

Used to refer to the action to sign into either a user or bot account, as opposed to the *login* process. Likewise, "sign out" is used to signify that the authorization should stop being valid.

## TL

### Type Language

File format used by Telegram to define all the types and requests available in a *layer*. Telegram's site has an [Overview of the TL language](#).

#### See also:

*Type Language brief*.

### yourself

The logged-in account, whether that represents a bot or a user with a phone number.

## API REFERENCE

This section contains all the functions and types offered by the library.

→ *Start reading Client API*

### 3.1 Client class

The *Client* class is the “entry point” of the library.

Most client methods have an alias in the respective types. For example, *Client.forward\_messages()* can also be invoked from *types.Message.forward()*. With a few exceptions, *client.verb\_object* methods also exist as *object.verb*.

```
class telethon.Client(session, api_id, api_hash=None, *, catch_up=False, check_all_handlers=False,  
flood_sleep_threshold=None, logger=None, update_queue_limit=None,  
device_model=None, system_version=None, app_version=None,  
system_lang_code=None, lang_code=None, datacenter=None, connector=None)
```

Bases: *object*

A client capable of connecting to Telegram and sending requests.

This class can be used as an asynchronous context manager to automatically *connect()* and *disconnect()*:

```
async with Client(session, api_id, api_hash) as client:  
    ... # automatically connect()-ed  
  
    ... # after exiting the block, disconnect() was automatically called
```

#### Parameters

- **session** (*str | Path | Storage | None*) – A name or path to a *.session* file, or a different storage.
- **api\_id** (*int*) – The API ID. See *Signing in* to learn how to obtain it.  
This is required to initialize the connection.
- **api\_hash** (*str | None*) – The API hash. See *Signing in* to learn how to obtain it.  
This is required to sign in, and can be omitted otherwise.
- **catch\_up** (*bool*) – Whether to “catch up” on updates that occurred while the client was not connected.  
If *True*, all updates that occurred while the client was offline will trigger your *event handlers*.

- **check\_all\_handlers** (*bool*) – Whether to always check all event handlers or stop early.

The library will call event handlers in the order they were added. By default, the library stops checking handlers as soon as a filter returns `True`.

By setting `check_all_handlers=True`, the library will keep calling handlers after the first match. Use `telethon.events.Continue` instead if you only want this behaviour sometimes.

- **flood\_sleep\_threshold** (*int* | *None*) – Maximum amount of time, in seconds, to automatically sleep before retrying a request. This sleeping occurs when `FLOOD_WAIT` (and similar) `RpcErrors` are raised by Telegram.
- **logger** (*Logger* | *None*) – Logger for the client. Any dependency of the client will use `logging.Logger.getChild()`. This effectively makes the parameter the root logger.

The default will get the logger for the package name from the root (usually `telethon`).

- **update\_queue\_limit** (*int* | *None*) – Maximum amount of updates to keep in memory before dropping them.

A warning will be logged on a cooldown if this limit is reached.

- **device\_model** (*str* | *None*) – Device model.
- **system\_version** (*str* | *None*) – System version.
- **app\_version** (*str* | *None*) – Application version.
- **system\_lang\_code** (*str* | *None*) – ISO 639-1 language code of the system’s language.
- **lang\_code** (*str* | *None*) – ISO 639-1 language code of the application’s language.
- **datacenter** (*DataCenter* | *None*) – Override the `data center` to connect to. Useful to connect to one of Telegram’s test servers.
- **connector** (*Connector* | *None*) – Asynchronous function called to connect to a remote address. By default, this is `asyncio.open_connection()`. In order to *use proxies*, you can set a custom connector.

See `Connector` for more details.

**add\_event\_handler**(*handler, event\_cls, filter=None*)

Register a callable to be invoked when the provided event type occurs.

#### Parameters

- **handler** (*Callable[[Event], Awaitable[Any]]*) – The callable to invoke when an event occurs. This is often just a function object.
- **event\_cls** (*Type[Event]*) – The event type to bind to the handler. When Telegram sends an update corresponding to this type, `handler` is called with an instance of this event type as the only argument.
- **filter** (*Callable[[Event], bool] | None*) – Filter function to call with the event before calling `handler`. If it returns `False`, `handler` will not be called. See the `filters` module to learn more.

#### Return type

None

### Example

```

async def my_print_handler(event):
    print(event.chat.name, event.text)

# Register a handler to be called on new messages
client.add_event_handler(my_print_handler, events.NewMessage)

# Register a handler to be called on new messages if they contain "hello" or "/"
↳start"
from telethon.events import filters

client.add_event_handler(
    my_print_handler,
    events.NewMessage,
    filters.Any(filters.Text(r'hello'), filters.Command('/start')),
)

```

#### See also:

`on()`, used to register handlers with the decorator syntax.

#### **async bot\_sign\_in**(token)

Sign in to a bot account.

##### Parameters

**token** (*str*) – The bot token obtained from `@BotFather`. It's a string composed of digits, a colon, and characters from the base-64 alphabet.

##### Returns

The bot user corresponding to *yourself*.

##### Return type

User

### Example

```

user = await client.bot_sign_in('12345:abc67DEF89ghi')
print('Signed in to bot account:', user.name)

```

**Caution:** Be sure to check `is_authorized()` before calling this function. Signing in often when you don't need to will lead to *RPC Errors*.

#### See also:

`request_login_code()`, used to sign in as a user instead.

#### **async check\_password**(token, password)

Check the two-factor-authentication (2FA) password. If it is correct, completes the login.

##### Parameters

- **token** (*PasswordToken*) – The return value from `sign_in()`.
- **password** (*str* / *bytes*) – The 2FA password.

**Returns**

The user corresponding to *yourself*.

**Return type**

User

**Example**

```
from telethon.types import PasswordToken

login_token = await client.request_login_code('+1 23 456')
password_token = await client.sign_in(login_token, input('code: '))
assert isinstance(password_token, PasswordToken)

user = await client.check_password(password_token, '1-L0V3+T313th0n')
print('Signed in to 2FA-protected account:', user.name)
```

**See also:**

`request_login_code()` and `sign_in()`

**async connect()**

Connect to the Telegram servers.

**Example**

```
await client.connect()
# success!
```

**Return type**

None

**property connected: bool**

True if `connect()` has been called previously.

This property will be set back to `False` after calling `disconnect()`.

This property does *not* check whether the connection is alive. The only way to check if the connection still works is to make a request.

**async delete\_dialog(chat)**

Delete a dialog.

This lets you leave a group, unsubscribe from a channel, or delete a one-to-one private conversation.

Note that the group or channel will not be deleted (other users will remain in it).

Note that bot accounts do not have dialogs, so this method will fail when used in a bot account.

**Parameters**

**chat** (Chat / PackedChat / int / str) – The *chat* representing the dialog to delete.

**Return type**

None



### Example

```

async for dialog in client.iter_dialogs():
    if 'dog pictures' in dialog.chat.name:
        # You've realized you're more of a cat person
        await client.delete_dialog(dialog.chat)

```

**async delete\_messages**(*chat*, *message\_ids*, \*, *revoke=True*)

Delete messages.

#### Parameters

- **chat** (*Chat* | *PackedChat* | *int* | *str*) – The *chat* where the messages are.

**Warning:** When deleting messages from private conversations or small groups, this parameter is currently ignored. This means the *message\_ids* may delete messages in different chats.

- **message\_ids** (*List[int]*) – The list of message identifiers to delete.
- **revoke** (*bool*) – When set to *True*, the message will be deleted for everyone that is part of *chat*. Otherwise, the message will only be deleted for *yourself*.

#### Returns

The amount of messages that were deleted.

#### Return type

*int*

### Example

```

# Delete two messages from chat for yourself
delete_count = await client.delete_messages(
    chat,
    [187481, 187482],
    revoke=False,
)
print('Deleted', delete_count, 'message(s)')

```

#### See also:

[telethon.types.Message.delete\(\)](#)

**async disconnect**()

Disconnect from the Telegram servers.

This call will only fail if saving the *session* fails.

### Example

```
await client.disconnect()
# success!
```

#### Return type

None

**async** `download(media, file)`

Download a file.

#### Parameters

- **media** (`File`) – The media file to download. This will often come from `telethon.types.Message.file`.
- **file** (`str` | `Path` | `OutFileLike`) – The output file path or file-like object. Note that the extension is not automatically added to the path. You can get the file extension with `telethon.types.File.ext`.

**Caution:** If the file already exists, it will be overwritten.

#### Return type

None

### Example

```
if photo := message.photo:
    await client.download(photo, f'picture{photo.ext}')

if video := message.video:
    with open(f'video{video.ext}', 'wb') as file:
        await client.download(video, file)
```

#### See also:

`get_file_bytes()`, for more control over the download.

**async** `edit_draft(chat, text=None, *, markdown=None, html=None, link_preview=False, reply_to=None)`

Set a draft message in a chat.

This can also be used to clear the draft by setting the text to an empty string "".

#### Parameters

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where the draft will be saved to.
- **text** (`str` | `None`) – See *Formatting messages*.
- **markdown** (`str` | `None`) – See *Formatting messages*.
- **html** (`str` | `None`) – See *Formatting messages*.
- **link\_preview** (`bool`) – See *Formatting messages*.
- **reply\_to** (`int` | `None`) – The message identifier of the message to reply to.

**Returns**

The created draft.

**Return type**

Draft

**Example**

```
# Set a draft with no formatting and print the date Telegram registered
draft = await client.edit_draft(chat, 'New text')
print('Set current draft on', draft.date)

# Set a draft using HTML formatting, with a reply, and enabling the link preview
await client.edit_draft(
    chat,
    html='Draft with <em>reply</em> an URL https://example.com',
    reply_to=message_id,
    link_preview=True
)
```

**See also:**

`telethon.types.Draft.edit()`

**async edit\_message**(*chat*, *message\_id*, \*, *text=None*, *markdown=None*, *html=None*, *link\_preview=False*, *buttons=None*)

Edit a message.

**Parameters**

- **chat** (*Chat* | *PackedChat* | *int* | *str*) – The *chat* where the message to edit is.
- **message\_id** (*int*) – The identifier of the message to edit.
- **text** (*str* | *None*) – See *Formatting messages*.
- **markdown** (*str* | *None*) – See *Formatting messages*.
- **html** (*str* | *None*) – See *Formatting messages*.
- **link\_preview** (*bool*) – See *Formatting messages*.
- **buttons** (*List[Button]* | *List[List[Button]]* | *None*) – The buttons to use for the message.

Only bot accounts can send buttons.

**Returns**

The edited message.

**Return type**

Message

### Example

```
# Edit message to have text without formatting
await client.edit_message(chat, msg_id, text='New text')

# Remove the link preview without changing the text
await client.edit_message(chat, msg_id, link_preview=False)
```

#### See also:

`telethon.types.Message.edit()`

**async forward\_messages**(*target, message\_ids, source*)

Forward messages from one *chat* to another.

#### Parameters

- **target** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where the messages will be forwarded to.
- **message\_ids** (`List[int]`) – The list of message identifiers to forward.
- **source** (`Chat` | `PackedChat` | `int` | `str`) – The source *chat* where the messages to forward exist.

#### Returns

The forwarded messages.

#### Return type

`List[Message]`

### Example

```
# Forward two messages from chat to the destination
messages = await client.forward_messages(
    destination,
    [187481, 187482],
    chat,
)
print('Forwarded', len(messages), 'message(s)')
```

#### See also:

`telethon.types.Message.forward()`

**get\_admin\_log**(*chat*)

Get the recent actions from the administrator’s log.

This method requires you to be an administrator in the *chat*.

The returned actions are also known as “admin log events”.

#### Parameters

**chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* to fetch recent actions from.

#### Returns

The recent actions.

#### Return type

`AsyncList[RecentAction]`

## Example

```

async for admin_log_event in client.get_admin_log(chat):
    if message := admin_log_event.deleted_message:
        print('Deleted:', message.text)

```

### **async** get\_chats(*chats*)

Get the latest basic information about the given chats.

This method is most commonly used to turn one or more *PackedChat* into the original *Chat*. This includes users, groups and broadcast channels.

#### Parameters

**chats** (*Sequence*[*Chat* | *PackedChat* | *int* | *str*]) – The users, groups or channels to fetch.

#### Returns

The fetched chats.

#### Return type

*List*[*Chat*]

## Example

```

# Retrieve a PackedChat from somewhere
packed_user = my_database.get_packed_winner()

# Fetch it
users = await client.get_chats([packed_user])
user = users[0] # user will be a User if our packed_user was a user

# Notify the user they won, using their current full name in the message
await client.send_message(packed_user, f'Congratulations {user.name}, you won!')

```

**Caution:** This method supports being called with anything that looks like a chat, like every other method. However, calling it with usernames or phone numbers will fetch the chats twice. If that's the case, consider using *resolve\_username()* or *get\_contacts()* instead.

### get\_contacts()

Get the users in your contact list.

#### Returns

Your contacts.

#### Return type

*AsyncList*[*User*]

### Example

```
async for user in client.get_contacts():
    print(user.name, user.id)
```

### get\_dialogs()

Get the dialogs you're part of.

This list includes the groups you've joined, channels you've subscribed to, and open one-to-one private conversations.

Note that bot accounts do not have dialogs, so this method will fail.

#### Returns

Your dialogs.

#### Return type

`AsyncList[Dialog]`

### Example

```
async for dialog in client.get_dialogs():
    print(
        dialog.chat.name,
        dialog.last_message.text if dialog.last_message else ''
    )
```

### get\_drafts()

Get all message drafts saved in any dialog.

#### Returns

The existing message drafts.

#### Return type

`AsyncList[Draft]`

### Example

```
# Clear all drafts
async for draft in client.get_drafts():
    await draft.delete()
```

### get\_file\_bytes(media)

Get the contents of an uploaded media file as chunks of `bytes`.

This lets you iterate over the chunks of a file and print progress while the download occurs.

If you just want to download a file to disk without printing progress, use `download()` instead.

#### Parameters

**media** (`File`) – The media file to download. This will often come from `telethon.types.Message.file`.

#### Return type

`AsyncList[bytes]`

### Example

```

if file := message.file:
    with open(f'media{file.ext}', 'wb') as fd:
        downloaded = 0
        async for chunk in client.get_file_bytes(file):
            downloaded += len(chunk)
            fd.write(chunk)
            print(f'Downloaded {downloaded // 1024}/{file.size // 1024} KiB')

```

### get\_handler\_filter(handler)

Get the filter associated to the given event handler.

#### Parameters

**handler** (*Callable*[[*Event*], *Awaitable*[*Any*]]) – The callable that was previously added as an event handler.

#### Returns

The filter, if *handler* was actually registered and had a filter.

#### Return type

*Callable*[[*Event*], bool] | None

### Example

```

from telethon.events import filters

# Get the current filter...
filt = client.get_handler_filter(my_handler)

# ...and "append" a new filter that also must match.
client.set_handler_filter(my_handler, filters.All(filt, filt.Text(r'test')))

```

### async get\_me()

Get information about *yourself*.

#### Returns

The user associated with the logged-in account, or *None* if the client is not authorized.

#### Return type

*User* | None

### Example

```

me = await client.get_me()
assert me is not None, "not logged in!"

if me.bot:
    print('I am a bot')

print('My name is', me.name)

```

(continues on next page)

```
if me.phone:
    print('My phone number is', me.phone)
```

**get\_messages**(*chat*, *limit=None*, \*, *offset\_id=None*, *offset\_date=None*)

Get the message history from a *chat*, from the newest message to the oldest.

The returned iterator can be `reversed()` to fetch from the first to the last instead.

#### Parameters

- **chat** (`Chat` / `PackedChat` / `int` / `str`) – The *chat* where the messages should be fetched from.
- **limit** (`int` / `None`) – How many messages to fetch at most.
- **offset\_id** (`int` / `None`) – Start getting messages with an identifier lower than this one. This means only messages older than the message with `id = offset_id` will be fetched.
- **offset\_date** (`datetime` / `None`) – Start getting messages with a date lower than this one. This means only messages sent before *offset\_date* will be fetched.

#### Returns

The message history.

#### Return type

`AsyncList[Message]`

### Example

```
# Get the last message in a chat
last_message = (await client.get_messages(chat, 1))[0]
print(message.sender.name, last_message.text)

# Print all messages before 2023 as HTML
from datetime import datetime

async for message in client.get_messages(chat, offset_date=datetime(2023, 1, 1),
    ←1)):
    print(message.sender.name, ':', message.html_text)

# Print the first 10 messages in a chat as markdown
async for message in reversed(client.get_messages(chat)):
    print(message.sender.name, ':', message.markdown_text)
```

**get\_messages\_with\_ids**(*chat*, *message\_ids*)

Get the full message objects from the corresponding message identifiers.

#### Parameters

- **chat** (`Chat` / `PackedChat` / `int` / `str`) – The *chat* where the message to fetch is.
- **message\_ids** (`List[int]`) – The message identifiers of the messages to fetch.

#### Returns

The matching messages. The order of the returned messages is *not* guaranteed to match the input. The method may return less messages than requested when some are missing.



**Return type**  
 AsyncList[Message]

### Example

```
# Get the first message (after "Channel created") of the chat
first_message = (await client.get_messages_with_ids(chat, [2]))[0]
```

### get\_participants(chat)

Get the participants in a group or channel, along with their permissions.

---

**Note:** Telegram is rather strict when it comes to fetching members. It is very likely that you will not be able to fetch all the members. There is no way to bypass this.

---

**Parameters**  
**chat** (Chat | PackedChat | int | str) – The *chat* to fetch participants from.

**Returns**  
 The participants.

**Return type**  
 AsyncList[Participant]

### Example

```
async for participant in client.get_participants(chat):
    print(participant.user.name)
```

### get\_profile\_photos(chat)

Get the profile pictures set in a chat, or user avatars.

**Parameters**  
**chat** (Chat | PackedChat | int | str) – The *chat* to fetch the profile photo files from.

**Returns**  
 The photo files.

**Return type**  
 AsyncList[File]

### Example

```
i = 0
async for photo in client.get_profile_photos(chat):
    await client.download(photo, f'{i}.jpg')
    i += 1
```

### async inline\_query(bot, query="", \*, chat=None)

Perform a @*bot inline query*.

It's known as inline because clients with a GUI display the results *inline*, after typing on the message input textbox, without sending any message.

**Parameters**

- **bot** (`Chat` / `PackedChat` / `int` / `str`) – The bot to sent the query string to.
- **query** (`str`) – The query string to send to the bot.
- **chat** (`Chat` / `PackedChat` / `int` / `str` / `None`) – Where the query is being made and will be sent. Some bots display different results based on the type of chat.

**Returns**

The query results returned by the bot.

**Return type**

`AsyncIterator[InlineResult]`

**Example**

```
i = 0

# This is equivalent to typing "@bot songs" in an official client
async for result in client.inline_query(bot, 'songs'):
    if 'keyword' in result.title:
        await result.send(chat)
        break

i += 1
if i == 10:
    break # did not find 'keyword' in the first few results
```

**async interactive\_login**(`phone_or_token=None`, \*, `password=None`)

Begin an interactive login if needed. If the account was already logged-in, this method simply returns *yourself*.

**Parameters**

- **phone\_or\_token** (`str` / `None`) – Bypass the phone number or bot token prompt, and use this value instead.
- **password** (`str` / `None`) – Bypass the 2FA password prompt, and use this value instead.

**Returns**

The user corresponding to *yourself*.

**Return type**

`User`

**Example**

```
# Interactive login from the terminal
me = await client.interactive_login()
print('Logged in as:', me.name)

# Automatic login to a bot account
await client.interactive_login('54321:hJrIQtVBab0M2Yqg4HL1K-EubfY_v2fEVR')
```

**See also:**

In-depth explanation for *Signing in*.

**async is\_authorized()**

Check whether the client instance is authorized (i.e. logged-in).

**Returns**

`True` if the client instance has signed-in.

**Return type**

`bool`

**Example**

```
if not await client.is_authorized():
    ... # need to sign in
```

**See also:**

`get_me()` can be used to fetch up-to-date information about *yourself* and check if you're logged-in at the same time.

**on(event\_cls, filter=None)**

Register the decorated function to be invoked when the provided event type occurs.

**Parameters**

- **event\_cls** (`Type[Event]`) – The event type to bind to the handler. When Telegram sends an update corresponding to this type, the decorated function is called with an instance of this event type as the only argument.
- **filter** (`Callable[[Event], bool] | None`) – Filter function to call with the event before calling *handler*. If it returns `False`, *handler* will not be called. See the *filters* module to learn more.

**Returns**

The decorator.

**Return type**

`Callable[[Callable[[Event], Awaitable[Any]]], Callable[[Event], Awaitable[Any]]]`

**Example**

```
# Register a handler to be called on new messages
@client.on(events.NewMessage)
async def my_print_handler(event):
    print(event.chat.name, event.text)

# Register a handler to be called on new messages if they contain "hello" or "/"
↳ start"
from telethon.events.filters import Any, Text, Command

@client.on(events.NewMessage, Any(Text(r'hello'), Command('/start')))
async def my_other_print_handler(event):
    print(event.chat.name, event.text)
```

**See also:**

`add_event_handler()`, used to register existing functions as event handlers.

**async pin\_message(chat, message\_id)**

Pin a message to be at the top.

**Parameters**

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where the message to pin is.
- **message\_id** (`int`) – The identifier of the message to pin.

**Returns**

The service message announcing the pin.

**Return type**

`Message`

**Example**

```
# Pin a message, then delete the service message
message = await client.pin_message(chat, 187481)
await message.delete()
```

**prepare\_album()**

Prepare an album upload to send.

Albums are a way to send multiple photos or videos as separate messages with the same grouped identifier.

**Returns**

A new album builder instance, with no media added to it yet.

**Return type**

`AlbumBuilder`

**Example**

```
# Prepare a new album
album = await client.prepare_album()

# Add a bunch of photos
for photo in ('a.jpg', 'b.png'):
    await album.add_photo(photo)
# A video in-between
await album.add_video('c.mp4')
# And another photo
await album.add_photo('d.jpeg')

# Album is ready to be sent to as many chats as needed
await album.send(chat)
```

**async read\_message(chat, message\_id)**

Mark messages as read.

This will send a read acknowledgment to all messages with identifiers below and up-to the given message identifier.

This is often represented as a blue double-check (✓✓).

A single check (✓) in Telegram often indicates the message was sent and perhaps received, but not read.

A clock (⌚) in Telegram often indicates the message was not yet sent at all. This most commonly occurs when sending messages without a network connection.

#### Parameters

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The chat where the messages to be marked as read are.
- **message\_id** (`int` | `Literal['all']`) – The identifier of the message to mark as read. All messages older (sent before) this one will also be marked as read.

The literal 'all' may be used to mark all messages in a chat as read.

#### Return type

None

#### Example

```
# Mark all messages as read
await client.read_message(chat, 'all')
```

#### `remove_event_handler(handler)`

Remove the handler as a function to be called when events occur. This is simply the opposite of `add_event_handler()`. Does nothing if the handler was not actually registered.

#### Parameters

**handler** (`Callable[[Event], Awaitable[Any]]`) – The callable to stop invoking when events occur.

#### Return type

None

#### Example

```
# Register a handler that removes itself when it receives 'stop'
@client.on(events.NewMessage)
async def my_handler(event):
    if 'stop' in event.text:
        client.remove_event_handler(my_handler)
    else:
        print('still going!')
```

#### `async request_login_code(phone)`

Request Telegram to send a login code to the provided phone number.

#### Parameters

**phone** (`str`) – The phone number string, in international format. The plus-sign + can be kept in the string.

#### Returns

Information about the sent code.

#### Return type

`LoginToken`

### Example

```
login_token = await client.request_login_code('+1 23 456...')
print(login_token.timeout, 'seconds before code expires')
```

**Caution:** Be sure to check `is_authorized()` before calling this function. Signing in often when you don't need to will lead to *RPC Errors*.

### See also:

`sign_in()`, to complete the login procedure.

### `async resolve_username(username)`

Resolve a username into a *chat*.

This method is rather expensive to call. It is recommended to use it once and then `types.Chat.pack()` the result. The packed chat can then be used (and re-fetched) more cheaply.

#### Parameters

**username** (*str*) – The public “@username” to resolve.

#### Returns

The matching chat.

#### Return type

*Chat*

### Example

```
print(await client.resolve_username('@cat'))
```

### `async run_until_disconnected()`

Keep running the library until a disconnection occurs.

Connection errors will be raised from this method if they occur.

#### Return type

*None*

### `search_all_messages(limit=None, *, query=None, offset_id=None, offset_date=None)`

Perform a global message search. This is used to search messages in no particular chat (i.e. everywhere possible).

#### Parameters

- **chat** – The *chat* where the message to edit is.
- **limit** (*int* | *None*) – How many messages to fetch at most.
- **query** (*str* | *None*) – Text query to use for fuzzy matching messages. The rules for how “fuzzy” works are an implementation detail of the server.
- **offset\_id** (*int* | *None*) – Start getting messages with an identifier lower than this one. This means only messages older than the message with `id = offset_id` will be fetched.
- **offset\_date** (*datetime* | *None*) – Start getting messages with a date lower than this one. This means only messages sent before `offset_date` will be fetched.

**Returns**

The found messages.

**Return type**

`AsyncList[Message]`

**Example**

```
async for message in client.search_all_messages(query='hello'):
    print(message.text)
```

**search\_messages**(*chat*, *limit=None*, \*, *query=None*, *offset\_id=None*, *offset\_date=None*)

Search messages in a chat.

**Parameters**

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where messages will be searched.
- **limit** (`int` | `None`) – How many messages to fetch at most.
- **query** (`str` | `None`) – Text query to use for fuzzy matching messages. The rules for how “fuzzy” works are an implementation detail of the server.
- **offset\_id** (`int` | `None`) – Start getting messages with an identifier lower than this one. This means only messages older than the message with `id = offset_id` will be fetched.
- **offset\_date** (`datetime` | `None`) – Start getting messages with a date lower than this one. This means only messages sent before *offset\_date* will be fetched.

**Returns**

The found messages.

**Return type**

`AsyncList[Message]`

**Example**

```
async for message in client.search_messages(chat, query='hello'):
    print(message.text)
```

**async send\_audio**(*chat*, *file*, *mime\_type=None*, \*, *size=None*, *name=None*, *duration=None*, *voice=False*, *title=None*, *performer=None*, *caption=None*, *caption\_markdown=None*, *caption\_html=None*, *reply\_to=None*, *buttons=None*)

Send an audio file.

Unlike `send_file()`, this method will attempt to guess the values for duration, title and performer if they are not provided.

**Parameters**

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where the audio media will be sent to.
- **file** (`str` | `Path` | `InFileLike` | `File`) – See `send_file()`.
- **size** (`int` | `None`) – See `send_file()`.
- **name** (`str` | `None`) – See `send_file()`.

- **mime\_type** (*str* | *None*) – See `send_file()`.
- **duration** (*float* | *None*) – See `send_file()`.
- **voice** (*bool*) – See `send_file()`.
- **title** (*str* | *None*) – See `send_file()`.
- **performer** (*str* | *None*) – See `send_file()`.
- **caption** (*str* | *None*) – See *Formatting messages*.
- **caption\_markdown** (*str* | *None*) – See *Formatting messages*.
- **caption\_html** (*str* | *None*) – See *Formatting messages*.
- **reply\_to** (*int* | *None*) –
- **buttons** (*List*[*Button*] | *List*[*List*[*Button*]] | *None*) –

**Return type**  
Message

### Example

```
await client.send_audio(chat, 'file.ogg', voice=True)
```

**async send\_file**(*chat*, *file*, \*, *size=None*, *name=None*, *mime\_type=None*, *compress=False*, *animated=False*, *duration=None*, *voice=False*, *title=None*, *performer=None*, *emoji=None*, *emoji\_sticker=None*, *width=None*, *height=None*, *round=False*, *supports\_streaming=False*, *muted=False*, *caption=None*, *caption\_markdown=None*, *caption\_html=None*, *reply\_to=None*, *buttons=None*)

Send any type of file with any amount of attributes.

This method will *not* attempt to guess any of the file metadata such as width, duration, title, etc. If you want to let the library attempt to guess the file metadata, use the type-specific methods to send media: `send_photo`, `send_audio` or `send_file`.

Unlike `send_photo()`, image files will be sent as documents by default.

### Parameters

- **chat** (*Chat* | *PackedChat* | *int* | *str*) – The *chat* where the message will be sent to.
- **path** – A local file path or *File* to send.
- **file** (*str* | *Path* | *InFileLike* | *File*) – The file to send.

This can be a path, relative or absolute, to a local file, as either a *str* or `pathlib.Path`.

It can also be a file opened for reading in binary mode, with its `read` method optionally being *async*. Note that the file descriptor will *not* be seeked back to the start before sending it.

If you wrote to an in-memory file, you probably want to `file.seek(0)` first. If you want to send *bytes*, wrap them in `io.BytesIO` first.

You can also pass any *File* that was previously sent in Telegram to send a copy. This will not download and re-upload the file, but will instead reuse the original without forwarding it.



Last, a URL can also be specified. For the library to detect it as a URL, the string *must* start with either `http://`` or ```https://``. Telethon will *not* download and upload the file, but will instead pass the URL to Telegram. If Telegram is unable to access the media, is too large, or is invalid, the method will fail.

When using URLs, it is recommended to explicitly pass either a name or define the mime-type. To make sure the URL is interpreted as an image, use `send_photo`.

- **size** (*int* / *None*) – The size of the local file to send.

This parameter **must** be specified when sending a previously-opened or in-memory files. The library will not seek the file to attempt to determine the size.

This can be less than the real file size, in which case only `size` bytes will be sent. This can be useful if you have a single buffer with multiple files.

- **name** (*str* / *None*) – Override for the default file name.

When given a string or path, its `name` will be used by default only if this parameter is omitted.

When given a *file-like object*, if it has a `.name` *str* property, it will be used. This is the case for files opened via `open()`.

This parameter **must** be specified when sending any other previously-opened or in-memory files.

- **mime\_type** (*str* / *None*) – Override for the default mime-type.

By default, the library will use `mimetypes.guess_type()` on the name.

If no mime-type is registered for the name's extension, `application/octet-stream` will be used.

- **compress** (*bool*) – Whether the image file is allowed to be compressed by Telegram.

If not, image files will be sent as document.

- **animated** (*bool*) – Whether the sticker is animated (not a static image).

- **duration** (*float* / *None*) – Duration, in seconds, of the audio or video.

This field should be specified when sending audios or videos from local files.

The floating-point value will be rounded to an integer.

- **voice** (*bool*) – Whether the audio is a live recording, often recorded right before sending it.

- **title** (*str* / *None*) – Title of the song in the audio file.

- **performer** (*str* / *None*) – Artist or main performer of the song in the audio file.

- **emoji** (*str* / *None*) – Alternative text for the sticker.

- **width** (*int* / *None*) – Width, in pixels, of the image or video.

This field should be specified when sending images or videos from local files.

- **height** (*int* / *None*) – Height, in pixels, of the image or video.

This field should be specified when sending images or videos from local files.

- **round** (*bool*) – Whether the video should be displayed as a round video.

- **supports\_streaming** (*bool*) – Whether clients are allowed to stream the video having to wait for a full download.

Note that the file format of the video must have streaming support.

- **muted** (*bool*) – Whether the sound of the video is or should be missing.

This is often used for short animations or “GIFs”.

- **caption** (*str* | *None*) – See *Formatting messages*.
- **caption\_markdown** (*str* | *None*) – See *Formatting messages*.
- **caption\_html** (*str* | *None*) – See *Formatting messages*.
- **emoji\_sticker** (*str* | *None*) –
- **reply\_to** (*int* | *None*) –
- **buttons** (*List*[*Button*] | *List*[*List*[*Button*]] | *None*) –

#### Return type

*Message*

#### Example

```
await client.send_file(chat, 'picture.jpg')

# Sending in-memory bytes
import io
data = b'my in-memory document'
await client.send_file(chat, io.BytesIO(data), size=len(data), name='doc.txt')
```

**async send\_message**(*chat*, *text=None*, \*, *markdown=None*, *html=None*, *link\_preview=False*, *reply\_to=None*, *buttons=None*)

Send a message.

#### Parameters

- **chat** (*Chat* | *PackedChat* | *int* | *str*) – The *chat* where the message will be sent to.
- **text** (*str* | *Message* | *None*) – See *Formatting messages*.
- **markdown** (*str* | *None*) – See *Formatting messages*.
- **html** (*str* | *None*) – See *Formatting messages*.
- **link\_preview** (*bool*) – See *Formatting messages*.
- **reply\_to** (*int* | *None*) – The message identifier of the message to reply to.
- **buttons** (*List*[*Button*] | *List*[*List*[*Button*]] | *None*) – The buttons to use for the message.

Only bot accounts can send buttons.

#### Return type

*Message*

## Example

```
await client.send_message(chat, markdown='**Hello!**')
```

```
async def send_photo(chat, file, *, size=None, name=None, mime_type=None, compress=True, width=None,
                    height=None, caption=None, caption_markdown=None, caption_html=None,
                    reply_to=None, buttons=None)
```

Send a photo file.

By default, the server will be allowed to *compress* the image. Only compressed images can be displayed as photos in applications. If *compress* is set to `False`, the image will be sent as a file document.

Unlike `send_file()`, this method will attempt to guess the values for width and height if they are not provided.

### Parameters

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where the photo media will be sent to.
- **file** (`str` | `Path` | `InFileLike` | `File`) – See `send_file()`.
- **size** (`int` | `None`) – See `send_file()`.
- **name** (`str` | `None`) – See `send_file()`.
- **mime\_type** (`str` | `None`) – See `send_file()`.
- **compress** (`bool`) – See `send_file()`.
- **width** (`int` | `None`) – See `send_file()`.
- **height** (`int` | `None`) – See `send_file()`.
- **caption** (`str` | `None`) – See *Formatting messages*.
- **caption\_markdown** (`str` | `None`) – See *Formatting messages*.
- **caption\_html** (`str` | `None`) – See *Formatting messages*.
- **reply\_to** (`int` | `None`) –
- **buttons** (`List[Button]` | `List[List[Button]]` | `None`) –

### Return type

`Message`

## Example

```
await client.send_photo(chat, 'photo.jpg', caption='Check this out!')
```

```
async def send_video(chat, file, *, size=None, name=None, mime_type=None, duration=None, width=None,
                   height=None, round=False, supports_streaming=False, muted=False, caption=None,
                   caption_markdown=None, caption_html=None, reply_to=None, buttons=None)
```

Send a video file.

Unlike `send_file()`, this method will attempt to guess the values for duration, width and height if they are not provided.

### Parameters

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where the message will be sent to.
- **file** (`str` | `Path` | `InFileLike` | `File`) – See `send_file()`.
- **size** (`int` | `None`) – See `send_file()`.
- **name** (`str` | `None`) – See `send_file()`.
- **mime\_type** (`str` | `None`) – See `send_file()`.
- **duration** (`float` | `None`) – See `send_file()`.
- **width** (`int` | `None`) – See `send_file()`.
- **height** (`int` | `None`) – See `send_file()`.
- **round** (`bool`) – See `send_file()`.
- **supports\_streaming** (`bool`) – See `send_file()`.
- **caption** (`str` | `None`) – See *Formatting messages*.
- **caption\_markdown** (`str` | `None`) – See *Formatting messages*.
- **caption\_html** (`str` | `None`) – See *Formatting messages*.
- **muted** (`bool`) –
- **reply\_to** (`int` | `None`) –
- **buttons** (`List[Button]` | `List[List[Button]]` | `None`) –

**Return type**

Message

**Example**

```
await client.send_video(chat, 'video.mp4', caption_markdown='*I cannot believe_↪this just happened*')
```

**async set\_chat\_default\_restrictions**(*chat*, *restrictions*, \*, *until=None*)

Set the default restrictions to apply to all participant in a chat.

**Parameters**

- **chat** (`Chat` | `PackedChat` | `int` | `str`) – The *chat* where the restrictions will be applied.
- **restrictions** (`Sequence[ChatRestriction]`) – The sequence of restrictions to apply.
- **until** (`datetime` | `None`) – Date until which the restrictions should be applied. By default, restrictions apply for as long as possible.

**Return type**

None

## Example

```

from datetime import datetime, timedelta
from telethon.types import ChatRestriction

# Don't allow anyone except administrators to send stickers for a day
await client.set_chat_default_restrictions(
    chat, user, [ChatRestriction.SEND_STICKERS],
    until=datetime.now() + timedelta(days=1))

# Remove all default restrictions from the chat
await client.set_chat_default_restrictions(chat, user, [])

```

See also:

`telethon.types.Group.set_default_restrictions()`

**set\_handler\_filter**(*handler*, *filter=None*)

Set the filter to use for the given event handler.

### Parameters

- **handler** (`Callable[[Event], Awaitable[Any]]`) – The callable that was previously added as an event handler.
- **filter** (`Callable[[Event], bool] | None`) – The filter to use for *handler*, or `None` to remove the old filter.

### Return type

None

## Example

```

from telethon.events import filters

# Change the filter to handle '/stop'
client.set_handler_filter(my_handler, filters.Command('/stop'))

# Remove the filter
client.set_handler_filter(my_handler, None)

```

**async set\_participant\_admin\_rights**(*chat*, *user*, *rights*)

Set the administrator rights granted to the participant in the chat.

If an empty sequence of rights is given, the user will be demoted and stop being an administrator.

In small group chats, there are no separate administrator rights. In this case, granting any right will make the user an administrator with all rights.

### Parameters

- **chat** (`Chat | PackedChat | int | str`) – The *chat* where the rights will be granted.
- **participant** – The participant to promote to administrator, usually a `types.User`.
- **rights** (`Sequence[AdminRight]`) – The sequence of rights to grant. Can be empty to revoke the administrator status from the participant.
- **user** (`Chat | PackedChat | int | str`) –

**Return type**

None

**Example**

```
from telethon.types import AdminRight

# Make user an administrator allowed to pin messages
await client.set_participant_admin_rights(
    chat, user, [AdminRight.PIN_MESSAGES])

# Demote an administrator
await client.set_participant_admin_rights(chat, user, [])
```

**See also:**

*telethon.types.Participant.set\_admin\_rights()*

**async set\_participant\_restrictions**(*chat, user, restrictions, \*, until=None*)

Set the restrictions to apply to a participant in the chat.

Restricting the participant to *VIEW\_MESSAGES* will kick them out of the chat.

In small group chats, there are no separate restrictions. In this case, any restriction will kick the participant. The participant's history will be revoked if the restriction to *VIEW\_MESSAGES* is applied.

**Parameters**

- **chat** (*Chat* / *PackedChat* / *int* / *str*) – The *chat* where the restrictions will be applied.
- **participant** – The participant to restrict or ban, usually a *types.User*.
- **restrictions** (*Sequence[ChatRestriction]*) – The sequence of restrictions to apply. Can be empty to remove all restrictions from the participant and unban them.
- **until** (*datetime* / *None*) – Date until which the restrictions should be applied. By default, restrictions apply for as long as possible.
- **user** (*Chat* / *PackedChat* / *int* / *str*) –

**Return type**

None

**Example**

```
from datetime import datetime, timedelta
from telethon.types import ChatRestriction

# Kick the user out of the chat
await client.set_participant_restrictions(
    chat, user, [ChatRestriction.VIEW_MESSAGES])

# Don't allow the user to send media for 5 minutes
await client.set_participant_restrictions(
    chat, user, [ChatRestriction.SEND_MEDIA],
```

(continues on next page)

(continued from previous page)

```

until=datetime.now() + timedelta(minutes=5))

# Unban the user
await client.set_participant_restrictions(chat, user, [])

```

**See also:**

`telethon.types.Participant.set_restrictions()`

**async sign\_in(token, code)**

Sign in to a user account.

**Parameters**

- **token** (`LoginToken`) – The login token returned from `request_login_code()`.
- **code** (`str`) – The login code sent by Telegram to a previously-authorized device. This should be a short string of digits.

**Returns**

The user corresponding to *yourself*, or a password token if the account has 2FA enabled.

**Return type**

User | PasswordToken

**Example**

```

from telethon.types import PasswordToken

login_token = await client.request_login_code('+1 23 456')
user_or_token = await client.sign_in(login_token, input('code: '))

if isinstance(password_token, PasswordToken):
    user = await client.check_password(password_token, '1-L0V3+T3l3th0n')

```

**See also:**

`check_password()`, the next step if the account has 2FA enabled.

**async sign\_out()**

Sign out, revoking the authorization of the current *session*.

**Example**

```

await client.sign_out() # turn off the lights
await client.disconnect() # shut the door

```

**Return type**

None

**async unpin\_message(chat, message\_id)**

Unpin one or all messages from the top.

**Parameters**

- **chat** (`Chat` / `PackedChat` / `int` / `str`) – The *chat* where the message pinned message is.
- **message\_id** (`int` / `Literal['all']`) – The identifier of the message to unpin, or 'all' to unpin them all.

**Return type**

None

**Example**

```
# Unpin all messages
await client.unpin_message(chat, 'all')
```

## 3.2 Events and filters

### 3.2.1 Events

Classes related to the different event types that wrap incoming Telegram updates.

**See also:**

The *Updates* concept to learn how to listen to these events.

**class telethon.events.ButtonCallback**

Bases: *Event*

Occurs when the user *click()*s a *Callback* button.

Only bot accounts can receive this event, because only bots can send *Callback* buttons.

**async answer**(*text=None*, *alert=False*)

Answer the callback query.

---

**Important:** You must call this function for the loading circle to stop on the user's side.

---

**Parameters**

- **text** (`str` / `None`) – The text of the message to display to the user, usually as a toast.
- **alert** (`bool`) – If `True`, the message will be shown as a pop-up alert that must be dismissed by the user.

**Return type**

None

**property data:** `bytes`

**async get\_message()**

Get the *Message* containing the button that was clicked.

If the message is too old and is no longer accessible, `None` is returned instead.



**Return type**

Message | None

**class telethon.events.Continue**Bases: `object`This is **not** an event type you can listen to.This is a sentinel value used to signal that the library should *Continue* calling other handlers.You can `return` this from your handlers if you want handlers registered after to also run.

The primary use case is having asynchronous filters inside your handler:

```

from telethon import events

@client.on(events.NewMessage)
async def admin_only_handler(event):
    allowed = await database.is_user_admin(event.sender.id)
    if not allowed:
        # this user is not allowed, fall-through the handlers
        return events.Continue

@client.on(events.NewMessage)
async def everyone_else_handler(event):
    ... # runs if admin_only_handler was not allowed

```

**class telethon.events.Event**Bases: `object`

The base type of all events.

**property client:** `Client`The `Client` that received this update.**class telethon.events.InlineQuery**Bases: `Event`Occurs when users type `@bot query` in their chat box.

Only bot accounts can receive this event.

**class telethon.events.MessageDeleted**Bases: `Event`

Occurs when one or more messages are deleted.

---

**Note:** Telegram does not send the contents of the deleted messages. Because they are deleted, it's also impossible to fetch them.

The chat is only known when the deletion occurs in broadcast channels or supergroups.

---

**property channel\_id:** `int | None`

The channel identifier of the supergroup or broadcast channel where the messages were deleted.

This will be `None` if the messages were deleted anywhere else.

**property message\_ids:** `List[int]`

The message identifiers of the messages that were deleted.

**class telethon.events.MessageEdited**

Bases: `Event`, `Message`

Occurs when a new message is sent or received.

This event can be treated as the `Message` itself.

**class telethon.events.MessageRead**

Bases: `Event`

Occurs both when your messages are read by others, and when you read messages.

**property chat:** `Chat`

The `chat` when the messages were read.

**property max\_message\_id\_read:** `int`

The highest message identifier of the messages that have been marked as read.

In other words, messages with an identifier below or equal (`<=`) to this value are considered read. Messages with an identifier higher (`>`) to this value are considered unread.

### Example

```
if message.id <= event.max_message_id_read:
    print('message is marked as read')
else:
    print('message is not yet marked as read')
```

**class telethon.events.NewMessage**

Bases: `Event`, `Message`

Occurs when a new message is sent or received.

This event can be treated as the `Message` itself.

**Caution:** Messages sent with the `Client` are also caught, so be careful not to enter infinite loops! This is true for all event types, including edits.

### 3.2.2 Filters

Filters are functions that accept a single parameter, an *Event* instance, and return a *bool*.

When the return value is *True*, the associated *events* handler will be invoked.

#### See also:

The *Updates* concept to learn to combine filters or define your own.

**class** telethon.events.filters.**All**(*filter1*, *filter2*, \**filters*)

Bases: *Combinable*

Combine multiple filters, returning *True* if all of the filters pass.

When either filter is *Combinable*, you can use the *&* operator instead.

```
from telethon.events.filters import All, Command, Text

@bot.on(events.NewMessage, All(Command('/start'), Text(r'data:\w+')))
async def handler(event): ...

# equivalent to:

@bot.on(events.NewMessage, Command('/start') & Text(r'data:\w+'))
async def handler(event): ...
```

#### Parameters

- **filter1** (*Callable*[[*Event*], *bool*]) – The first filter to check.
- **filter2** (*Callable*[[*Event*], *bool*]) – The second filter to check.
- **filters** (*Callable*[[*Event*], *bool*]) – The rest of filters to check.

**property filters:** *Tuple*[*Callable*[[*Event*], *bool*], ...]

The filters being checked, in order.

**class** telethon.events.filters.**Any**(*filter1*, *filter2*, \**filters*)

Bases: *Combinable*

Combine multiple filters, returning *True* if any of the filters pass.

When either filter is *Combinable*, you can use the *|* operator instead.

```
from telethon.events.filters import Any, Command

@bot.on(events.NewMessage, Any(Command('/start'), Command('/help')))
async def handler(event): ...

# equivalent to:

@bot.on(events.NewMessage, Command('/start') | Command('/help'))
async def handler(event): ...
```

#### Parameters

- **filter1** (*Callable*[[*Event*], *bool*]) – The first filter to check.
- **filter2** (*Callable*[[*Event*], *bool*]) – The second filter to check if the first one failed.

- **filters** (*Callable*[[*Event*], *bool*]) – The rest of filters to check if the first and second one failed.

**property filters:** *Tuple*[*Callable*[[*Event*], *bool*], ...]

The filters being checked, in order.

**class** telethon.events.filters.**ChatType**(*type*)

Bases: *Combinable*

Filter by chat type using *isinstance*().

**Parameters**

**type** (*Type*[*User* | *Group* | *Channel*]) – The chat type to filter on.

### Example

```
from telethon import events
from telethon.events import filters
from telethon.types import Channel

# Handle only messages from broadcast channels
@client.on(events.NewMessage, filters.ChatType(Channel))
async def handler(event):
    print(event.text)
```

**property type:** *Type*[*User* | *Group* | *Channel*]

The chat type this filter is filtering on.

**class** telethon.events.filters.**Chats**(*chat\_ids*)

Bases: *Combinable*

Filter by *event.chat.id*, if the event has a chat.

**Parameters**

**chat\_ids** (*Sequence*[*int*]) – The chat identifiers to filter on.

**property chat\_ids:** *Set*[*int*]

A copy of the set of chat identifiers this filter is filtering on.

**class** telethon.events.filters.**Command**(*command*)

Bases: *Combinable*

Filter by *event.text* to make sure the first word matches the command or the command + '@' + username, using the username of the logged-in account.

For example, if the logged-in account has an username of "bot", then the filter *Command('/help')* will match both *"/help"* and *"/help@bot"*, but not *"/list"* or *"/help@other"*.

**Parameters**

**command** (*str*) – The command to match on.

---

**Note:** The leading forward-slash is not automatically added! This allows for using a different prefix or no prefix at all.

---

---

**Note:** The username is taken from the `session` to avoid network calls. If a custom storage returns the incorrect username, the filter will misbehave. If there is no username, then the `"/help@other"` syntax will be ignored.

---

**class** telethon.events.filters.**Data**(*data*)

Bases: *Combinable*

Filter by `event.data` using a full bytes match, used for events such as `telethon.events.ButtonCallback`.

It checks if `event.data` is equal to the data passed to the filter.

**Parameters**

**data** (*bytes*) – Bytes to match data with.

**class** telethon.events.filters.**Forward**

Bases: *Combinable*

Filter by `event.forward_info`, that is, messages that have been forwarded from elsewhere.

**class** telethon.events.filters.**Incoming**

Bases: *Combinable*

Filter by `event.incoming`, that is, messages sent from others to the logged-in account.

**class** telethon.events.filters.**Media**(\**types*)

Bases: *Combinable*

Filter by the media type in the message.

By default, this filter will pass if the message has any media.

Note that link previews are only considered media if they have a photo or document.

When you specify one or more media types, *only* those types will be considered.

You can use literal strings or the constants defined by the filter.

**Parameters**

**types** (*Literal*['photo', 'audio', 'video']) – The media types to filter on. This is all of them if none are specified.

**AUDIO** = 'audio'

**PHOTO** = 'photo'

**VIDEO** = 'video'

**property types:** *Tuple*[*Literal*['photo', 'audio', 'video'], ...]

The media types being checked.

**class** telethon.events.filters.**Not**(*filter*)

Bases: *Combinable*

Negate the output of a single filter, returning `True` if the nested filter does *not* pass.

When the filter is *Combinable*, you can use the `~` operator instead.

```
from telethon.events.filters import All, Command

@bot.on(events.NewMessage, Not(Command('/start')))
async def handler(event): ...
```

(continues on next page)

(continued from previous page)

```
# equivalent to:
@bot.on(events.NewMessage, ~Command('/start'))
async def handler(event): ...
```

**Parameters**

**filter** (*Callable*[[*Event*], *bool*]) – The filter to negate.

**property filter:** *Callable*[[*Event*], *bool*]

The filter being negated.

**class telethon.events.filters.Outgoing**

Bases: *Combinable*

Filter by `event.outgoing`, that is, messages sent from the logged-in account.

This is not a reliable way to check that the update was not produced by the logged-in account in broadcast channels.

**class telethon.events.filters.Reply**

Bases: *Combinable*

Filter by `event.replied_message_id`, that is, messages which are a reply to another message.

**class telethon.events.filters.Senders**(*sender\_ids*)

Bases: *Combinable*

Filter by `event.sender.id`, if the event has a sender.

**Parameters**

**sender\_ids** (*Sequence*[*int*]) – The sender identifiers to filter on.

**property sender\_ids:** *Set*[*int*]

A copy of the set of sender identifiers this filter is filtering on.

**class telethon.events.filters.Text**(*regex*)

Bases: *Combinable*

Filter by `event.text` using a *regular expression* pattern.

The pattern is searched on the text anywhere, not matched at the start. Use the '^' anchor if you want to match the text from the start.

The match, if any, is discarded. If you need to access captured groups, you need to manually perform the check inside the handler instead.

Note that the caption text in messages with media is also searched. If you want to filter based on media, use *Media*.

**Parameters**

**regex** (*Union*[*str*, *re.Pattern*[*str*]]) – The regular expression to `re.search()` with on the text.

## 3.3 Types

This section contains most custom types used by the library. *Events and filters* and the *Client class* get their own section to prevent the page from growing out of control.

Some of these are further divided into additional submodules. This keeps them neatly grouped and avoids polluting a single module too much.

### 3.3.1 Core types

Classes for the various objects the library returns.

```
class telethon.types.AdminRight(value, names=None, *, module=None, qualname=None, type=None,
                                start=1, boundary=None)
```

Bases: `Enum`

A right that can be granted to a chat's administrator.

---

**Note:** The specific values of the enumeration are not covered by `semver`. They also may do nothing in future updates if Telegram decides to change them.

---

**BAN\_USERS** = 'ban\_users'

Allows setting the banned rights of other users in a group or channel.

**CHANGE\_INFO** = 'change\_info'

Allows editing the description in a group or channel.

**DELETE\_MESSAGES** = 'delete\_messages'

Allows deleting messages in a group or channel.

**DELETE\_STORIES** = 'delete\_stories'

Allows deleting stories in a channel.

**EDIT\_MESSAGES** = 'edit\_messages'

Allows editing messages in a group or channel.

**EDIT\_STORIES** = 'edit\_stories'

Allows editing stories in a channel.

**INVITE\_USERS** = 'invite\_users'

Allows inviting other users to the group or channel.

**MANAGE\_ADMINS** = 'add\_admins'

Allows setting the same or less administrator rights to other users in the group or channel.

**MANAGE\_CALLS** = 'manage\_call'

Allows managing group or channel calls.

**MANAGE\_TOPICS** = 'manage\_topics'

Allows managing the topics in a group.

**OTHER** = 'other'

Unspecified.

**PIN\_MESSAGES** = 'pin\_messages'

Allows pinning a message to the group or channel.

**POST\_MESSAGES** = 'post\_messages'

Allows sending messages in a broadcast channel.

**POST\_STORIES** = 'post\_stories'

Allows posting stories in a channel.

**REMAIN\_ANONYMOUS** = 'anonymous'

Allows the administrator to remain anonymous.

**class** telethon.types.**AlbumBuilder**

Bases: `object`

Album builder to prepare albums with multiple files before sending it all at once.

This class is constructed by calling `telethon.Client.prepare_album()`.

**async** `add_photo`(*file*, \*, *size=None*, *caption=None*, *caption\_markdown=None*, *caption\_html=None*)

Add a photo to the album.

#### Parameters

- **file** (*str* | *Path* | *InFileLike*) – The photo to attach to the album.  
This behaves the same way as the file parameter in `telethon.Client.send_file()`, *except* that it cannot be previously-sent media.
- **size** (*int* | *None*) – See `telethon.Client.send_file()`.
- **caption** (*str* | *None*) – See *Formatting messages*.
- **caption\_markdown** (*str* | *None*) – See *Formatting messages*.
- **caption\_html** (*str* | *None*) – See *Formatting messages*.

#### Return type

None

**async** `add_video`(*file*, \*, *size=None*, *name=None*, *mime\_type=None*, *duration=None*, *width=None*, *height=None*, *round=False*, *supports\_streaming=False*, *muted=False*, *caption=None*, *caption\_markdown=None*, *caption\_html=None*)

Add a video to the album.

#### Parameters

- **file** (*str* | *Path* | *InFileLike*) – The video to attach to the album.  
This behaves the same way as the file parameter in `telethon.Client.send_file()`, *except* that it cannot be previously-sent media.
- **size** (*int* | *None*) – See `telethon.Client.send_file()`.
- **name** (*str* | *None*) – See `telethon.Client.send_file()`.
- **mime\_type** (*str* | *None*) – See `telethon.Client.send_file()`.
- **duration** (*float* | *None*) – See `telethon.Client.send_file()`.
- **width** (*int* | *None*) – See `telethon.Client.send_file()`.
- **height** (*int* | *None*) – See `telethon.Client.send_file()`.



- **round** (*bool*) – See *telethon.Client.send\_file()*.
- **supports\_streaming** (*bool*) – See *telethon.Client.send\_file()*.
- **muted** (*bool*) – See *telethon.Client.send\_file()*.
- **caption** (*str* | *None*) – See *Formatting messages*.
- **caption\_markdown** (*str* | *None*) – See *Formatting messages*.
- **caption\_html** (*str* | *None*) – See *Formatting messages*.

**Return type**

None

**async send**(*chat*, \*, *reply\_to=None*)

Send the album.

**Returns**

All sent messages that are part of the album.

**Parameters**

- **chat** (*Chat* | *PackedChat* | *int* | *str*) –
- **reply\_to** (*int* | *None*) –

**Return type***List*[*Message*]**Example**

```
album = await client.prepare_album()
for photo in ('a.jpg', 'b.png'):
    await album.add_photo(photo)

messages = await album.send(chat)
```

**class telethon.types.AsyncList**Bases: *ABC*, *Generic*[*T*]

An asynchronous list.

It can be awaited to get all the items as a normal *list*, or iterated over via *async for*.

Both approaches will perform as many requests as needed to retrieve the items, but awaiting will need to do it all at once, which can be slow.

Using asynchronous iteration will perform the requests lazily as needed, and lets you break out of the loop at any time to stop fetching items.

The *len()* of the asynchronous list will be the “total count” reported by the server. It does not necessarily reflect how many items will actually be returned. This count can change as more items are fetched. Note that this method cannot be awaited.

## Example

`telethon.Client.get_messages()` returns an `AsyncList[Message]`. This means:

```
# You can await it directly:
messages = await client.get_messages(chat, 1)
# ...and now messages is a normal list with a single Message.

# Or you can use async for:
async for message in client.get_messages(chat, 1):
    ... # the messages are fetched lazily, rather than all up-front.
```

**class** telethon.types.Button(*text*)

Bases: `object`

The button base type.

All other *buttons* inherit this class.

You can only click buttons that have been received from Telegram. Attempting to click a button you created will fail with an error.

Not all buttons can be clicked, and each button will do something different when clicked. The reason for this is that Telethon cannot interact with any user to complete certain tasks. Only straightforward actions can be performed automatically, such as sending a text message.

To check if a button is clickable, use `hasattr()` on the 'click' method.

### Parameters

**text** (*str*) – See below.

**property text:** `str`

The button's text that is displayed to the user.

**class** telethon.types.CallbackAnswer

Bases: `object`

A bot's *Callback answer()*.

**property text:** `str | None`

The answer's text, usually displayed as a toast.

**property url:** `str | None`

The answer's URL.

**class** telethon.types.Channel

Bases: `Chat`

A broadcast channel.

You can get a channel from messages via `telethon.types.Message.chat`, or from methods such as `telethon.Client.resolve_username()`.

**property id: int**

The chat's integer identifier.

This identifier is always a positive number.

This property is always present.

**property name: str**

The channel's title.

This property is always present, but may be the empty string.

**pack()**

Pack the chat into a compact and reusable object.

This object can be easily serialized and saved to persistent storage. Unlike resolving usernames, packed chats can be reused without costly calls.

**See also:**

*Chats*

**Return type**

PackedChat | None

**property username: str | None**

The primary *@username* of the chat.

The returned string will *not* contain the at-sign @.

**class telethon.types.Chat**

Bases: ABC

The base class for all chat types.

This will either be a *User*, *Group* or *Channel*.

**abstract property id: int**

The chat's integer identifier.

This identifier is always a positive number.

This property is always present.

**abstract property name: str**

The full name of the user, group or channel.

For users, this will be the *User.first\_name* concatenated with the *User.last\_name*.

For groups and channels, this will be their title.

If there is no name (such as for deleted accounts), an empty string '' will be returned.

**abstract pack()**

Pack the chat into a compact and reusable object.

This object can be easily serialized and saved to persistent storage. Unlike resolving usernames, packed chats can be reused without costly calls.

**See also:**

*Chats*

**Return type**

PackedChat | None

**abstract property username:** str | NoneThe primary `@username` of the chat.The returned string will *not* contain the at-sign `@`.**class** telethon.types.ChatRestriction(*value, names=None, \*, module=None, qualname=None, type=None, start=1, boundary=None*)

Bases: Enum

A restriction that may be applied to a banned chat's participant.

A banned participant is completely banned from a chat if they are forbidden to `VIEW_MESSAGES`.A banned participant that can `VIEW_MESSAGES` is restricted, but can still be part of the chat.

---

**Note:** The specific values of the enumeration are not covered by `semver`. They also may do nothing in future updates if Telegram decides to change them.

---

**CHANGE\_INFO = 'change\_info'**

Prevents changing the description of the chat.

**EMBED\_LINKS = 'embed\_links'**

Prevents sending messages that include links to external URLs to the chat.

**INVITE\_USERS = 'invite\_users'**

Prevents inviting users to the chat.

**MANAGE\_TOPICS = 'manage\_topics'**

Prevents managing the topics of the chat.

**PIN\_MESSAGES = 'pin\_messages'**

Prevents pinning messages to the chat.

**SEND\_AUDIOS = 'send\_audios'**

Prevents sending audio media files to the chat.

**SEND\_DOCUMENTS = 'send\_docs'**

Prevents sending document media files to the chat.

**SEND\_GAMES = 'send\_games'**Prevents sending `@bot inline` games to the chat.**SEND\_GIFS = 'send\_gifs'**

Prevents sending muted looping video media (“GIFs”) to the chat.

**SEND\_INLINE = 'send\_inline'**Prevents sending messages via `@bot inline` to the chat.**SEND\_MEDIA = 'send\_media'**

Prevents sending messages with media such as photos or documents to the chat.

**SEND\_MESSAGES = 'send\_messages'**

Prevents sending messages to the chat.

**SEND\_PHOTOS = 'send\_photos'**

Prevents sending photo media files to the chat.

**SEND\_PLAIN\_MESSAGES = 'send\_plain'**

Prevents sending plain text messages with no media to the chat.

**SEND\_POLLS = 'send\_polls'**

Prevents sending poll media to the chat.

**SEND\_ROUND\_VIDEOS = 'send\_roundvideos'**

Prevents sending round video media files to the chat.

**SEND\_STICKERS = 'send\_stickers'**

Prevents sending sticker media to the chat.

**SEND\_VIDEOS = 'send\_videos'**

Prevents sending video media files to the chat.

**SEND\_VOICE\_NOTES = 'send.voices'**

Prevents sending voice note audio media files to the chat.

**VIEW\_MESSAGES = 'view\_messages'**

Prevents being in the chat and fetching the message history.

Applying this restriction will kick the participant out of the group.

**class telethon.types.Dialog**

Bases: `object`

A dialog.

This represents an open conversation your chat list.

This includes the groups you've joined, channels you've subscribed to, and open one-to-one private conversations.

You can obtain dialogs with methods such as `telethon.Client.get_dialogs()`.

**property chat:** `Chat`

The chat where messages are sent in this dialog.

**property draft:** `Draft | None`

The message draft within this dialog, if any.

This property does not update when the draft changes.

**property latest\_message:** `Message | None`

The latest message sent or received in this dialog, if any.

This property does not update when new messages arrive.

**property unread\_count:** `int`

The amount of unread messages in this dialog.

This property does not update when messages are read or sent.

**class telethon.types.Draft**

Bases: `object`

A draft message in a chat.

You can obtain drafts with methods such as `telethon.Client.get_drafts()`.

**property chat:** `Chat`

The chat where the draft is saved.

This is also the chat where the message will be sent to by `send()`.

**property date:** `datetime | None`

The date when the draft was last updated.

**async delete()**

Clear the contents of this draft to delete it.

**Return type**

`None`

**async edit**(`text=None, *, markdown=None, html=None, link_preview=False, reply_to=None`)

Replace the current draft with a new one.

**Parameters**

- **text** (`str | None`) – See *Formatting messages*.
- **markdown** (`str | None`) – See *Formatting messages*.
- **html** (`str | None`) – See *Formatting messages*.
- **link\_preview** (`bool`) – See *Formatting messages*.
- **reply\_to** (`int | None`) – The message identifier of the message to reply to.

**Returns**

The edited draft.

**Return type**

`Draft`

### Example

```
new_draft = await old_draft.edit('new text', link_preview=False)
```

**property link\_preview:** `bool`

`True` if the link preview is allowed to exist when sending the message.

**property replied\_message\_id:** `int | None`

Get the message identifier of message this draft will reply to once sent.

**async send()**

Send the contents of this draft to the chat.

The draft will be cleared after being sent.

**Returns**

The sent message.

**Return type**

`Message`

## Example

```
await draft.send(clear=False)
```

**property text:** `str | None`

The `text` of the message that will be sent.

**property text\_html:** `str | None`

The `text_html` of the message that will be sent.

**property text\_markdown:** `str | None`

The `text_markdown` of the message that will be sent.

**class telethon.types.File**

Bases: `object`

File information of media sent to Telegram that can be downloaded.

You can get a file from messages via `telethon.types.Message.file`, or from methods such as `telethon.Client.get_profile_photos()`.

**async download**(*file*)

Alias for `telethon.Client.download()`.

**Parameters**

**file** (`str | Path | OutFileLike`) – See `download()`.

**Return type**

None

**property ext:** `str`

The file extension, including the leading dot ..

If the name is not known, the mime-type is used in `mimetypes.guess_extension()`.

If no extension is known for the mime-type, the empty string will be returned. This makes it safe to always append this property to a file name.

**property height:** `int | None`

The width of the image or video, if available.

**property name:** `str | None`

The file name, if known.

**property thumbnails:** `List[File]`

The file thumbnails.

For photos, these are often downscaled versions of the original size.

For documents, these will be the thumbnails present in the document.

**property width:** `int | None`

The width of the image or video, if available.

**class telethon.types.Group**

Bases: `Chat`

A small group or supergroup.

You can get a group from messages via `telethon.types.Message.chat`, or from methods such as `telethon.Client.resolve_username()`.

**property id: int**

The chat's integer identifier.

This identifier is always a positive number.

This property is always present.

**property is\_megagroup: bool**

Whether the group is a supergroup.

These are known as “megagroups” in Telegram’s API, and are different from “gigagroups”.

**property name: str**

The group's title.

This property is always present, but may be the empty string.

**pack()**

Pack the chat into a compact and reusable object.

This object can be easily serialized and saved to persistent storage. Unlike resolving usernames, packed chats can be reused without costly calls.

**See also:**

*Chats*

**Return type**

`PackedChat` | `None`

**async set\_default\_restrictions(restrictions, \*, until=None)**

Alias for `telethon.Client.set_chat_default_restrictions()`.

**Parameters**

- **restrictions** (`Sequence[ChatRestriction]`) –
- **until** (`datetime` | `None`) –

**Return type**

`None`

**property username: str | None**

The primary `@username` of the chat.

The returned string will *not* contain the at-sign @.

**class telethon.types.InlineButton(text)**

Bases: `Button`

Inline button base type.

Inline buttons appear directly under a message (inline in the chat history).

You cannot create a naked `InlineButton` directly. Instead, it can be used to check whether a button is inline or not.

Buttons that behave as a “custom key” and replace the user’s virtual keyboard can be tested by checking that they are not inline.



## Example

```
from telethon.types import buttons

is_inline_button = isinstance(button, buttons.Inline)
is_keyboard_button = not isinstance(button, buttons.Inline)
```

### Parameters

**text** (*str*) – See below.

## class telethon.types.InlineResult

Bases: `object`

A single inline result from an inline query made to a bot.

This is returned when calling `telethon.Client.inline_query()`.

### property description: `str | None`

The description of the result, if available.

### async send(*chat=None*)

Send the result to the desired chat.

#### Parameters

**chat** (`Chat | PackedChat | int | str | None`) – The chat where the inline result should be sent to.

This can be omitted if a chat was previously specified in the `inline_query()`.

#### Returns

The sent message.

#### Return type

`Message`

### property title: `str`

The title of the result, or the empty string if there is none.

### property type: `str`

## class telethon.types.LoginToken

Bases: `object`

Result of requesting a login code via `telethon.Client.request_login_code()`.

### property timeout: `int | None`

Number of seconds before this token expires.

This property does not return different values as the current time advances. To determine when the token expires, add the timeout to the current time as soon as the token is obtained.

## class telethon.types.Message

Bases: `object`

A sent message.

You can get a message from `telethon.events.NewMessage`, or from methods such as `telethon.Client.get_messages()`.

Empty messages can occur very rarely when fetching the message history. In these cases, only the `id` and `:attr`peer`` properties are guaranteed to be present. To determine whether a message is empty, its truthy value can be checked via `object.__bool__()`:

```
async for message in client.iter_messages(chat):
    if not message:
        print('Found empty message with ID', message.id)
```

**property audio:** `File` | `None`

The audio media `file` in the message.

This can also be used as a way to check that the message media is an audio.

**property buttons:** `List[List[Button]]` | `None`

The buttons attached to the message.

These are displayed under the message if they are `InlineButton`, and replace the user's virtual keyboard otherwise.

The returned value is a list of rows, each row having a list of buttons, one per column. The amount of columns in each row can vary. For example:

```
buttons = [
    [col_0,      col_1], # row 0
    [   col_0   ], # row 1
    [col_0, col_1, col_2], # row 2
]

row = 2
col = 1
button = buttons[row][col] # the middle button on the bottom row
```

**property can\_forward:** `bool`

**property chat:** `Chat`

The `chat` when the message was sent.

**property date:** `datetime` | `None`

The date when the message was sent.

**async delete**(`*`, `revoke=True`)

Alias for `telethon.Client.delete_messages()`.

**Parameters**

**revoke** (`bool`) – See `delete_messages()`.

**Return type**

`None`

**async edit**(`text=None`, `markdown=None`, `html=None`, `link_preview=False`, `buttons=None`)

Alias for `telethon.Client.edit_message()`.

**Parameters**

- **text** (*str* | *None*) – See *Formatting messages*.
- **markdown** (*str* | *None*) – See *Formatting messages*.
- **html** (*str* | *None*) – See *Formatting messages*.
- **link\_preview** (*bool*) – See *send\_message()*.
- **buttons** (*List[Button]* | *List[List[Button]]* | *None*) – See *send\_message()*.

**Return type**

Message

**property file:** *File* | *None*

The downloadable file in the message.

This might also come from a link preview.

Unlike *photo*, *audio* and *video*, this property does not care about the media type, only whether it can be downloaded.

This means the file will be *None* for other media types, such as polls, venues or contacts.

**async forward**(*target*)

Alias for *telethon.Client.forward\_messages()*.

**Parameters**

**target** (*Chat* | *PackedChat* | *int* | *str*) – See *forward\_messages()*.

**Return type**

Message

**property forward\_info:** *None***async get\_replied\_message**()

Alias for *telethon.Client.get\_messages\_with\_ids()*.

If all you want is to check whether this message is a reply, use *replied\_message\_id*.

**Return type**Message | *None***property grouped\_id:** *int* | *None*

If the message is grouped with others in an album, return the group identifier.

Messages with the same *grouped\_id* will belong to the same album.

Note that there can be messages in-between that do not have a *grouped\_id*.

**property id:** *int*

The message identifier.

**See also:**

*Messages*, which contains an in-depth explanation of message counters.

**property incoming:** *bool*

*True* if the message is incoming. This would mean another user sent it, and the currently logged-in user received it.

This is usually the opposite of *outgoing*, although some messages can be neither.

**property link\_preview:** *None*

**property outgoing:** `bool`

`True` if the message is outgoing. This would mean the currently logged-in user sent it.

This is usually the opposite of *incoming*, although some messages can be neither.

**property photo:** `File | None`

The compressed photo media *file* in the message.

This can also be used as a way to check that the message media is a photo.

**async pin()**

Alias for `telethon.Client.pin_message()`.

**Return type**

`Message`

**async read()**

Alias for `telethon.Client.read_message()`.

**Return type**

`None`

**property replied\_message\_id:** `int | None`

Get the message identifier of the replied message.

**See also:**

`get_replied_message()`

**async reply**(`text=None`, \*, `markdown=None`, `html=None`, `link_preview=False`, `buttons=None`)

Alias for `telethon.Client.send_message()` with the `reply_to` parameter set to this message.

**Parameters**

- `text` (`str | Message | None`) – See *Formatting messages*.
- `markdown` (`str | None`) – See *Formatting messages*.
- `html` (`str | None`) – See *Formatting messages*.
- `link_preview` (`bool`) – See `send_message()`.
- `buttons` (`List[Button] | List[List[Button]] | None`) – See `send_message()`.

**Return type**

`Message`

**async respond**(`text=None`, \*, `markdown=None`, `html=None`, `link_preview=False`, `buttons=None`)

Alias for `telethon.Client.send_message()`.

**Parameters**

- `text` (`str | Message | None`) – See *Formatting messages*.
- `markdown` (`str | None`) – See *Formatting messages*.
- `html` (`str | None`) – See *Formatting messages*.
- `link_preview` (`bool`) – See `send_message()`.
- `buttons` (`List[Button] | List[List[Button]] | None`) – See `send_message()`.

**Return type**

`Message`

**property sender:** `Chat | None`

The *chat* that sent the message.

This will usually be a *User*, but can also be a *Channel*.

If there is no sender, it means the message was sent by an anonymous user.

**property silent:** `bool`

`True` if the message is silent and should not cause a notification.

**property text:** `str | None`

The message text without any formatting.

**property text\_html:** `str | None`

The message text formatted using standard *HTML* elements.

See *Formatting messages* to learn the *HTML* elements used.

**property text\_markdown:** `str | None`

The message text formatted as *CommonMark*'s markdown.

See *Formatting messages* to learn the formatting characters used.

**async unpin()**

Alias for `telethon.Client.unpin_message()`.

**Return type**

`None`

**property video:** `File | None`

The video media *file* in the message.

This can also be used as a way to check that the message media is a video.

**class telethon.types.PackedChat**(*ty, id, access\_hash*)

Bases: `object`

A compact representation of a *chat*.

You can reuse it as many times as you want.

You can call `chat.pack()` on *User*, *Group* or *Channel* to obtain it.

**See also:**

*Chats*

**Parameters**

- `ty` (`PackedType`) –
- `id` (`int`) –
- `access_hash` (`int | None`) –

`access_hash`

**classmethod from\_bytes**(*data*)

**Parameters**

`data` (`bytes`) –

**Return type**

*Self*

**classmethod** `from_hex(hex)`

Convenience method to convert hexadecimal numbers into bytes then passed to `from_bytes()`:

**Parameters**

**hex** (*str*) – Hexadecimal numbers to convert from.

**Return type**

*Self*

```
assert PackedChat.from_hex(packed.hex) == packed
```

**property** `hex: str`

Convenience property to convert to bytes and represent them as hexadecimal numbers:

```
assert packed.hex == bytes(packed).hex()
```

**id**

**is\_channel()**

**Return type**

*bool*

**is\_chat()**

**Return type**

*bool*

**is\_user()**

**Return type**

*bool*

**ty**

**class** `telethon.types.PackedType`(*value, names=None, \*, module=None, qualname=None, type=None, start=1, boundary=None*)

Bases: `IntFlag`

The type of a `PackedChat`.

**BOT** = 3

**BROADCAST** = 48

**CHAT** = 4

**GIGAGROUP** = 56

**MEGAGROUP** = 40

**USER** = 2

**class** `telethon.types.Participant`

Bases: `object`

A participant in a chat, including the corresponding user and permissions.

You can obtain participants with methods such as `telethon.Client.get_participants()`.

**property admin\_rights:** `Set[AdminRight] | None`

The set of administrator rights this participant has been granted, if they are an administrator.

**property banned:** `Chat | None`

The banned participant.

This will usually be a `User`.

**property creator:** `bool`

`True` if the participant is the creator of the chat.

**property left:** `Chat | None`

The participant that has left the group.

This will usually be a `User`.

**property restrictions:** `Set[ChatRestriction] | None`

The set of restrictions applied to this participant, if they are banned.

**async set\_admin\_rights**(*rights*)

Alias for `telethon.Client.set_participant_admin_rights()`.

**Parameters**

**rights** (`Sequence[AdminRight]`) –

**Return type**

`None`

**async set\_restrictions**(*restrictions*, \*, *until=None*)

Alias for `telethon.Client.set_participant_restrictions()`.

**Parameters**

- **restrictions** (`Sequence[ChatRestriction]`) –
- **until** (`datetime | None`) –

**Return type**

`None`

**property user:** `User | None`

The user participant that is currently present in the chat.

This will be `None` if the participant was instead `banned` or has `left`.

**class telethon.types.PasswordToken**

Bases: `object`

Result of attempting to `sign_in()` to a 2FA-protected account.

**property hint:** `str`

The password hint, or the empty string if none is known.

**class telethon.types.RecentAction**

Bases: `object`

A recent action in a chat, also known as an “admin log event action” or `ChannelAdminLogEvent`.

Only administrators of the chat can access these.

You can obtain recent actions with methods such as `telethon.Client.get_admin_log()`.

**property id: int**

The identifier of this action.

This identifier is *not* the same as the one in the message that was edited or deleted.

**class telethon.types.User**

Bases: *Chat*

A user, representing either a bot account or an account created with a phone number.

You can get a user from messages via *telethon.types.Message.sender*, or from methods such as *telethon.Client.resolve\_username()*.

**property bot: bool**

**property first\_name: str**

**property id: int**

The chat's integer identifier.

This identifier is always a positive number.

This property is always present.

**property last\_name: str**

**property name: str**

The user's full name.

This property joins both the *first\_name* and *last\_name* into a single string.

This property is always present, but may be the empty string.

**pack()**

Pack the chat into a compact and reusable object.

This object can be easily serialized and saved to persistent storage. Unlike resolving usernames, packed chats can be reused without costly calls.

**See also:**

*Chats*

**Return type**

*PackedChat* | None

**property phone: str | None**

**property username: str | None**

The primary *@username* of the chat.

The returned string will *not* contain the at-sign @.



### 3.3.2 Keyboard buttons

Keyboard buttons.

This includes both the buttons returned by `telethon.types.Message.buttons` and those you can define when using `telethon.Client.send_message()`:

```
from telethon.types import buttons

# As a user account, you can search for and click on buttons:
for row in message.buttons:
    for button in row:
        if isinstance(button, buttons.Callback) and button.data == b'data':
            await button.click()

# As a bot account, you can send them:
await bot.send_message(chat, text, buttons=[
    buttons.Callback('Demo', b'data')
])
```

**class** `telethon.types.buttons.Callback`(*text*, *data=None*)

Bases: `InlineButton`

Inline button that will trigger a `telethon.events.ButtonCallback` with the button's data.

#### Parameters

- **text** (*str*) – See below.
- **data** (*bytes* / *None*) – See below.

**async** `click()`

Click the button, sending the button's *data* to the bot.

The bot will receive a `ButtonCallback` event which they must quickly `answer()`.

The bot's answer will be returned, or `None` if they don't answer in time.

#### Return type

`CallbackAnswer` | `None`

**property data:** `bytes`

The button's binary payload.

This data will be received by `telethon.events.ButtonCallback` when the button is pressed.

**class** `telethon.types.buttons.RequestGeoLocation`(*text*)

Bases: `Button`

Keyboard button that will prompt the user to share the geo point with their current location.

#### Parameters

- **text** (*str*) – See below.

**class** `telethon.types.buttons.RequestPhone`(*text*)

Bases: `Button`

Keyboard button that will prompt the user to share the contact with their phone number.

#### Parameters

- **text** (*str*) – See below.

**class** telethon.types.buttons.**RequestPoll**(*text*, \*, *quiz=False*)

Bases: *Button*

Keyboard button that will prompt the user to create a poll.

**Parameters**

- **text** (*str*) – See below.
- **quiz** (*bool*) –

**class** telethon.types.buttons.**SwitchInline**(*text*, *query=None*)

Bases: *InlineButton*

Inline button that will switch the user to inline mode to trigger *telethon.events.InlineQuery*.

**Parameters**

- **text** (*str*) – See below.
- **query** (*str* | *None*) – See below.

**property query:** **str**

The query string to set by default on the user's message input.

**class** telethon.types.buttons.**Text**(*text*)

Bases: *Button*

This is the most basic keyboard button and only has *text*.

Note that it is not possible to distinguish between a *click()* to this button being and the user typing the text themselves.

**Parameters**

**text** (*str*) – See below.

**async click()**

Click the button, sending a message to the chat as-if the user typed and sent the text themselves.

**Return type**

*Message*

**property text:** **str**

The button's text that is both displayed to the user and will be sent on *click()*.

**class** telethon.types.buttons.**Url**(*text*, *url=None*)

Bases: *InlineButton*

Inline button that will prompt the user to open the specified URL when clicked.

**Parameters**

- **text** (*str*) – See below.
- **url** (*str* | *None*) – See below.

**property url:** **str**

The URL to open.

### 3.3.3 Errors

**class** telethon.**RpcError**(\* , code=0, name="", value=None, caused\_by=None)

Bases: [ValueError](#)

A Remote Procedure Call Error.

Only occurs when the answer to a request sent to Telegram is not the expected result. The library will never construct instances of this error by itself.

This is the parent class of all [telethon.errors](#) subtypes.

#### Parameters

- **code** (*int*) – See below.
- **name** (*str*) – See below.
- **value** (*int* | *None*) – See below.
- **caused\_by** (*int* | *None*) – Constructor identifier of the request that caused the error.

See also:

[RPC Errors](#)

**property code:** *int*

Integer code of the error.

This usually reassembles an [HTTP status code](#).

**property name:** *str*

Name of the error, usually in [SCREAMING\\_CASE](#).

**property value:** *int* | *None*

Integer value contained within the error.

For example, if the *name* is 'FLOOD\_WAIT', this would be the number of seconds.

**telethon.errors**

Factory-object returning subclasses of [RpcError](#).

You can think of it as a module with an infinite amount of error types in it.

When accessing any attribute in this object, a subclass of [RpcError](#) will be returned.

The returned type will catch [RpcError](#) if the [RpcError.name](#) matches the attribute converted to [SCREAMING\\_CASE](#).

For example:

```
from telethon import errors

try:
    await client.send_message(chat, text)
except errors.FloodWait as e:
    await asyncio.sleep(e.value)
```

Note how the [RpcError.value](#) field is still accessible, as it's a subclass of [RpcError](#). The code above is equivalent to the following:

```

from telethon import RpcError

try:
    await client.send_message(chat, text)
except RpcError as e:
    if e.name == 'FLOOD_WAIT':
        await asyncio.sleep(e.value)
    else:
        raise

```

This factory object is merely a convenience.

There is one exception, and that is when the attribute name starts with 'Code' and ends with a number:

```

try:
    await client.send_message(chat, text)
except errors.Code420:
    await asyncio.sleep(e.value)

```

The above snippet is equivalent to checking `RpcError.code` instead:

```

try:
    await client.send_message(chat, text)
except RpcError as e:
    if e.code == 420:
        await asyncio.sleep(e.value)
    else:
        raise

```

### 3.3.4 Private definitions

**Warning:** These are **not** intended to be imported directly. They are *not* available from `telethon.types`.

This section exists for documentation purposes only.

`telethon._impl.client.types.async_list.T`

Generic parameter used by `AsyncList`.

**class** `telethon._impl.client.events.filters.combinators.Combinable`

Bases: `ABC`

Subclass that enables filters to be combined.

- The `bitwise or` operator `|` can be used to combine filters with `Any`.
- The `bitwise and` operator `&` can be used to combine filters with `All`.
- The `bitwise invert` operator `~` can be used to negate a filter with `Not`.

Filters combined this way will be merged. This means multiple `|` or `&` will lead to a single `Any` or `All` being used. Multiple `~` will toggle between using `Not` and not using it.

**class** telethon.\_impl.client.types.file.InFileLike(\*args, \*\*kwargs)

Bases: Protocol

A file-like object used for input only. The `read()` method can be `async`.

This is never returned and should never be constructed. It's only used in function parameters.

**read**(*n*)

Read from the file or buffer.

**Parameters**

**n** (*int*) – Maximum amount of bytes that should be returned.

**Return type**

bytes | *Coroutine*[Any, Any, bytes]

**class** telethon.\_impl.client.types.file.OutFileLike(\*args, \*\*kwargs)

Bases: Protocol

A file-like object used for output only. The `write()` method can be `async`.

This is never returned and should never be constructed. It's only used in function parameters.

**write**(*data*)

Write all the data into the file or buffer.

**Parameters**

**data** (*bytes*) – Data that must be written to the buffer entirely.

**Return type**

Any | *Coroutine*[Any, Any, Any]

**class** telethon.\_impl.mtsender.sender.AsyncReader(\*args, \*\*kwargs)

Bases: Protocol

A `asyncio.StreamReader`-like class.

**async read**(*n*)

Must behave like `asyncio.StreamReader.read()`.

**Parameters**

**n** (*int*) – Amount of bytes to read at most.

**Return type**

bytes

**class** telethon.\_impl.mtsender.sender.AsyncWriter(\*args, \*\*kwargs)

Bases: Protocol

A `asyncio.StreamWriter`-like class.

**close**()

Must behave like `asyncio.StreamWriter.close()`.

**Return type**

None

**async drain**()

Must behave like `asyncio.StreamWriter.drain()`.

**Return type**

None

**async wait\_closed()**

Must behave like `asyncio.StreamWriter.wait_closed()`.

**Return type**

None

**write(data)**

Must behave like `asyncio.StreamWriter.write()`.

**Parameters**

**data** (*bytes*) – Data that must be entirely written or buffered until `drain()` is called.

**Return type**

None

**class telethon.\_impl.mtsender.sender.Connector(\*args, \*\*kwargs)**

Bases: `Protocol`

A *Connector* is any function that takes in the following two positional parameters as input:

- The ip address as a `str`. This might be either a IPv4 or IPv6.
- The port as a `int`. This will be a number below  $2^{16}$ , often 443.

and returns a `tuple[AsyncReader, AsyncWriter]`.

You can use a custom connector to connect to Telegram through proxies. The library will only ever open remote connections through this function.

The default connector is `asyncio.open_connection()`, defined as:

```
default_connector = lambda ip, port: asyncio.open_connection(ip, port)
```

If your connector needs additional parameters, you can use either the `lambda` syntax or `functools.partial()`.

**See also:**

The *Data centers* concept has examples on how to combine proxy libraries with Telethon.

## 3.4 Sessions

Classes related to session data and session storages.

---

**Note:** This module is mostly of interest if you plan to write a custom session storage. You should not need to interact with these types directly under normal use.

---

**See also:**

The *Sessions* concept for more details.

### 3.4.1 Storages

**class** telethon.session.Storage

Bases: ABC

Interface declaring the required methods of a *session* storage.

**abstract async delete()**

Delete the saved *Session*.

This method is called by the library post *log\_out*.

Note that both *load()* and *save()* may still be called after.

**Return type**

None

**abstract async load()**

Load the *Session* instance, if any.

This method is called by the library prior to *connect*.

**Returns**

The previously-saved session.

**Return type**

*Session* | None

**abstract async save(session)**

Save the *Session* instance to persistent storage.

This method is called by the library post *disconnect*.

**Parameters**

**session** (*Session*) – The session information that should be persisted.

**Return type**

None

**class** telethon.session.SQLiteSession(*file*)

Bases: *Storage*

Session storage backed by SQLite.

SQLite is a reliable way to persist data to disk and offers file locking.

Paths without extension will have  `'.session'` appended to them. This is by convention, and to make it harder to commit session files to an VCS by accident (adding `*.session` to `.gitignore` will catch them).

**Parameters**

**file** (*str* | *Path*) –

**async delete()**

Delete the saved *Session*.

This method is called by the library post *log\_out*.

Note that both *load()* and *save()* may still be called after.

**Return type**

None

**async load()**

Load the *Session* instance, if any.

This method is called by the library prior to *connect*.

**Returns**

The previously-saved session.

**Return type**

*Session* | None

**async save(session)**

Save the *Session* instance to persistent storage.

This method is called by the library post *disconnect*.

**Parameters**

**session** (*Session*) – The session information that should be persisted.

**Return type**

None

**class telethon.session.MemorySession(session=None)**

Bases: *Storage*

Session storage without persistence.

This is the simplest storage and is the one used by default.

Session data is only kept in memory and is not persisted to disk.

**Parameters**

**session** (*Session* | None) –

**async delete()**

Delete the saved *Session*.

This method is called by the library post *log\_out*.

Note that both *load()* and *save()* may still be called after.

**Return type**

None

**async load()**

Load the *Session* instance, if any.

This method is called by the library prior to *connect*.

**Returns**

The previously-saved session.

**Return type**

*Session* | None

**async save(session)**

Save the *Session* instance to persistent storage.

This method is called by the library post *disconnect*.

**Parameters**

**session** (*Session*) – The session information that should be persisted.

**Return type**

None



**session**

### 3.4.2 Types

**class** telethon.session.**Session**(\*, dcs=None, user=None, state=None)

Bases: `object`

A Telethon *session*.

A session instance contains the required information to login into your Telegram account. **Never** give the saved session file to anyone else or make it public.

Leaking the session file will grant a bad actor complete access to your account, including private conversations, groups you're part of and list of contacts (though not secret chats).

If you think the session has been compromised, immediately terminate all sessions through an official Telegram client to revoke the authorization.

#### Parameters

- **dcs** (`List[DataCenter]` / `None`) – See below.
- **user** (`User` / `None`) – See below.
- **state** (`UpdateState` / `None`) – See below.

**VERSION = 1**

Current version.

Will be incremented if new fields are added.

**dcs**

List of known data-centers.

**state**

Update state.

**user**

Information about the logged-in user.

**class** telethon.session.**DataCenter**(\*, id, ipv4\_addr=None, ipv6\_addr=None, auth=None)

Bases: `object`

Data-center information.

#### Parameters

- **id** (`int`) – See below.
- **ipv4\_addr** (`str` / `None`) – See below.
- **ipv6\_addr** (`str` / `None`) – See below.
- **auth** (`bytes` / `None`) – See below.

**auth**

Authentication key to encrypt communication with.

**id**

The DC identifier.

**ipv4\_addr**

The IPv4 socket server address of the DC, in 'ip:port' format.

**ipv6\_addr**

The IPv6 socket server address of the DC, in 'ip:port' format.

**class** telethon.session.**User**(\*, id, dc, bot, username)

Bases: `object`

Information about the logged-in user.

**Parameters**

- **id** (*int*) – See below.
- **dc** (*int*) – See below.
- **bot** (*bool*) – See below.
- **username** (*str* / *None*) – See below.

**bot**

`True` if the user is from a bot account.

**dc**

Data-center identifier of the user's "home" DC.

**id**

User identifier.

**username**

User's primary username.

**class** telethon.session.**UpdateState**(\*, pts, qts, date, seq, channels)

Bases: `object`

Update state for an account.

**Parameters**

- **pts** (*int*) – See below.
- **qts** (*int*) – See below.
- **date** (*int*) – See below.
- **seq** (*int*) – See below.
- **channels** (*List* [*ChannelState*]) – See below.

**channels**

Update state for channels.

**date**

Date of the latest update sequence.

**pts**

The primary partial sequence number.

**qts**

The secondary partial sequence number.

**seq**

The sequence number.

**class** telethon.session.ChannelState(\*, *id*, *pts*)

Bases: `object`

Update state for a channel.

**Parameters**

- **id** (*int*) – See below.
- **pts** (*int*) – See below.

**id**

The channel identifier.

**pts**

The channel's partial sequence number.



## DEVELOPMENT RESOURCES

Tips and tricks to develop both with the library and for the library.

→ *Start reading Changelog*

### 4.1 Changelog (Version History)

#### 4.1.1 v2.0-alpha.0

Scheme layer used: 165

[View new and changed raw API methods.](#)

---

**Important: The first alpha of Telethon v2 is here!**

---

What a ride!

This has been a long-time coming and I am so glad I don't need to work on the old and messy code-base anymore!

This changelog will no longer have catchy names for the different versions. It's been a while since I was able to come up with meaningful names, anyway.

Instead of breaking down what's new, changed, or removed from this version, there is a migration guide. Be warned, it's a lot! But in short, this is mostly a "reset" to cut down on the bloat of the library. The library should still result familiar.

**See also:**

*[Migrating from v1 to v2](#)*

Your v1 code likely won't be compatible with v2. However, a lot of effort has gone into making v2 type-safe. This means you can use static type-checkers to be confident you're using the new methods correctly.

A lot of work has been put into ensuring the documentation is the best it can be. If you find any errata or confusing sections, please [open a Documentation Issue](#).

### What happens to Telethon v1 now?

Version 1 will probably be maintained for a while longer.

New layers will be released, and bugs that make it unusable fixed. As long as I have the time to do so. This will not be a priority. There is no specific end-of-life date, but it will not last forever.

No new features will be added to v1, or smaller bugs fixed. This should not be news to anyone, as it has already been the case for a while now.

You can keep using v1 for now. But a lot of work has been put into v2 and its migration guide. I hope you can take this opportunity to migrate and dust off some of your old code!

### 4.1.2 v1.x and below

The old [v1 changelog](#) is still available online. And in the history of the repository. It has been removed from the current documentation to keep it lighter.

## 4.2 Migrating from v1 to v2

v2 is a complete reboot of Telethon v1. Because a lot of the library has suffered radical changes, there are no plans to provide “bridge” methods emulating the old interface. Doing so would take a lot of extra time and energy, and it’s honestly not fun.

What this means is that your v1 code very likely won’t run in v2. Sorry. I hope you can use this opportunity to shake up your dusty code into a cleaner design, too.

The common theme in v2 could be described as “no bullshit”.

v1 had grown a lot of features. A lot of them did a lot of things, at all once, in slightly different ways. Semver allows additions, so v2 will start out smaller and grow in a controlled manner.

Custom types were a frankenstein monster, combining both raw and manually-defined properties in hacky ways. Type hinting was an unmaintained disaster. Features such as file IDs, proxies and a lot of utilities were pretty much abandoned.

The several attempts at making v2 a reality over the years starting from the top did not work out. A bottom-up approach was needed. So a full rewrite was warranted.

TLSharp was Telethon’s seed. Telethon v0 was needed to learn Python at all. Telethon v1 was necessary to learn what was a good design, and what wasn’t. This inspired [grammers](#), a Rust re-implementation with a thought-out design. Telethon v2 completes the loop by porting [grammers](#) back to Python, now built with years of experience in the Telegram protocol.

It turns out static type checking is a very good idea for long-running projects. So I strongly encourage you to use [mypy](#) when developing code with Telethon v2. I can guarantee you will into far less problems.

Without further ado, let’s take a look at the biggest changes. This list may not be exhaustive, but it should give you an idea on what to expect. If you feel like a major change is missing, please [open an issue](#).

## 4.2.1 Complete project restructure

The public modules under the telethon now make actual sense.

- The root telethon package contains the basics like the *Client* and *RpcError*.
- *telethon.types* contains all the types, for your type-hinting needs.
- *telethon.events* contains all the events.
- *telethon.events.filters* contains all the event filters.
- *telethon.session* contains the session storages, should you choose to build a custom one.
- *telethon.errors* is no longer a module. It's actually a factory object returning new error types on demand. This means you don't need to wait for new library versions to be released to catch them.

---

**Note:** Be sure to check the documentation for *telethon.errors* to learn about error changes. Notably, errors such as *FloodWaitError* no longer have a *.seconds* field. Instead, every value for every error type is always *.value*.

---

This was also a good opportunity to remove a lot of modules that were not supposed to public in their entirety: *.crypto*, *.extensions*, *.network*, *.custom*, *.functions*, *.helpers*, *.hints*, *.password*, *.requestiter*, *.sync*, *.types*, *.utils*.

## 4.2.2 TelegramClient renamed to Client

You can rename it with `as` during import if you want to use the old name.

Python allows using namespaces via packages and modules. Therefore, the full name *telethon.Client* already indicates it's from telethon, so the old Telegram prefix was redundant.

## 4.2.3 No telethon.sync hack

You can no longer import `telethon.sync` to have most calls wrapped in `asyncio.loop.run_until_complete()` for you.

## 4.2.4 Raw API is now private

v2 aims to comply with *Semantic Versioning*. This is impossible because Telegram is a live service that can change things any time. But we can get pretty close.

In v1, minor version changes bumped Telegram's *layer*. This technically violated semver, because they were part of a public module.

To allow for new layers to be added without the need for major releases, `telethon._tl` is instead private. Here's the recommended way to import and use it now:

```
from telethon import _tl as tl

was_reset = await client(tl.functions.account.reset_wall_papers())

if isinstance(chat, tl.abcs.User):
    if isinstance(chat, tl.types.UserEmpty):
        return
    # chat is tl.types.User
```

There are three modules (four, if you count `core`, which you probably should not use). Each of them can have an additional namespace (as seen above with `account.`).

- `tl.functions` contains every *TL* definition treated as a function. The naming convention now follows Python's, and are `snake_case`.
- `tl.abcs` contains every abstract class, the “boxed” types from Telegram. You can use these for your type-hinting needs.
- `tl.types` contains concrete instances, the “bare” types Telegram actually returns. You'll probably use these with `isinstance()` a lot.

Most custom `types` will also have a private `_raw` attribute with the original value from Telegram.

### 4.2.5 Raw API has a reduced feature-set

The string representation is now on `object.__repr__()`, not `object.__str__()`.

All types use `__slots__` to save space. This means you can't add extra fields to these at runtime unless you subclass.

The `.stringify()` methods on all TL types no longer exists. Instead, you can use a library like `beauty-print`.

The `.to_dict()` method on all TL types no longer exists. The same is true for `.to_json()`. Instead, you can use a library like `json-pickle` or write your own:

```
def to_dict(obj):
    if obj is None or isinstance(obj, (bool, int, bytes, str)): return obj
    if isinstance(obj, list): return [to_dict(x) for x in obj]
    if isinstance(obj, dict): return {k: to_dict(v) for k, v in obj.items()}
    return {slot: to_dict(getattr(obj, slot)) for slot in obj.__slots__}
```

Lesser-known methods such as `TLObject.pretty_format`, `serialize_bytes`, `serialize_datetime` and `from_reader` are also gone. The remaining methods are:

- `Serializable.constructor_id()` class-method, to get the integer identifier of the corresponding type constructor.
- `Serializable.from_bytes()` class-method, to convert serialized `bytes` back into the class.
- `object.__bytes__()` instance-method, to serialize the instance into `bytes` the way Telegram expects.

Functions are no longer a class with attributes. They serialize the request immediately. This means you cannot create request instance and change it later. Consider using `functools.partial()` if you want to reuse parts of a request instead.

Functions no longer have an asynchronous `.resolve()`. This used to let you pass usernames and have them be resolved to `InputPeer` automatically (unless it was nested).

### 4.2.6 Changes to start and client context-manager

You can no longer `start()` the client.

Instead, you will need to first `connect()` and then start the `interactive_login()`.

In v1, the when using the client as a context-manager, `start()` was called. Since that method no longer exists, it now instead only `connect()` and `disconnect()`.

This means you won't get annoying prompts in your terminal if the session was not authorized. It also means you can now use the context manager even with custom login flows.



The old `sign_in()` method also sent the code, which was rather confusing. Instead, you must now `request_login_code()` as a separate operation.

The old `log_out()` was also renamed to `sign_out()` for consistency with `sign_in()`.

The old `is_user_authorized()` was renamed to `is_authorized()` since it works for bot accounts too.

## 4.2.7 Unified client iter and get methods

The client no longer has `client.iter_...` methods.

Instead, they return a type that supports both `await` and `async for`:

```
messages = await client.get_messages(chat, 100)
# or
async for message in client.get_messages(chat, 100):
    ...
```

**Note:** `Client.get_messages()` no longer has funny rules for the `limit` either. If you `await` it without `limit`, it will probably take a long time to complete. This is in contrast to v1, where `get` defaulted to 1 message and `iter` to no `limit`.

## 4.2.8 Removed client methods and properties

### No `client.parse_mode` property.

Instead, you need to specify how the message text should be interpreted every time. In `send_message()`, use `text=`, `markdown=` or `html=`. In `send_file()` and friends, use one of the `caption` parameters.

### No `client.loop` property.

Instead, you can use `asyncio.get_running_loop()`.

### No `client.conversation()` method.

Instead, you will need to [design your own FSM](#). The simplest approach could be using a global `states` dictionary storing the next function to call:

```
from functools import partial

states = {}

@client.on(events.NewMessage)
async def conversation_entry_point(event):
    if fn := state.get(event.sender.id):
        await fn(event)
    else:
        await event.respond('Hi! What is your name?')
        state[event.sender.id] = handle_name
```

(continues on next page)

(continued from previous page)

```
async def handle_name(event):
    await event.respond('What is your age?')
    states[event.sender.id] = partial(handle_age, name=event.text)

async def handle_age(event, name):
    age = event.text
    await event.respond(f'Hi {name}, I am {age} too!')
    del states[event.sender.id]
```

### No `client.kick_participant()` method.

This is not a thing in Telegram. It was implemented by restricting and then removing the restriction.

The old `client.edit_permissions()` was renamed to `Client.set_participant_restrictions()`. This defines the restrictions a banned participant has applied (bans them from doing those things). Revoking the right to view messages will kick them. This rename should avoid confusion, as it is now clear this is not to promote users to admin status.

For administrators, `client.edit_admin` was renamed to `Client.set_participant_admin_rights()` for consistency.

You can also use the aliases on the `Participant`, `types.Participant.set_restrictions()` and `types.Participant.set_admin_rights()`.

Note that a new method, `Client.set_chat_default_restrictions()`, must now be used to set a chat's default rights. You can also use the alias on the `Group`, `types.Group.set_default_restrictions()`.

### No `client.download_profile_photo()` method.

You can simply use `Client.download()` now. Note that `download()` no longer supports downloading contacts as `.vcard`.

### No `client.set_proxy()` method.

Proxy support is no longer built-in. They were never officially maintained. This doesn't mean you can't use them. You're now free to choose your own proxy library and pass a different connector to the `Client` constructor.

This should hopefully make it clear that most connection issues when using proxies do *not* come from Telethon.

### No `client.set_receive_updates` method.

It was not working as expected.

**No `client.catch_up()` method.**

You can still configure it when creating the *Client*, which was the only way to make it work anyway.

**No `client.action()` method.****No `client.takeout()` method.****No `client.qr_login()` method.****No `client.edit_2fa()` method.****No `client.get_stats()` method.****No `client.edit_folder()` method.****No `client.build_reply_markup()` method.****No `client.list_event_handlers()` method.**

These are out of scope for the time being. They might be re-introduced in the future if there is a burning need for them and are not difficult to maintain. This doesn't mean you can't do these things anymore though, since the *Raw API* is still available.

Telethon v2 is committed to not exposing the raw API under any public API of the `telethon` package. This means any method returning data from Telegram must have a custom wrapper object and be maintained too. Because the standards are higher, the barrier of entry for new additions and features is higher too.

## 4.2.9 Removed or renamed message properties and methods

Messages no longer have `raw_text` or `message` properties.

Instead, you can access the `types.Message.text`, `text_markdown` or `text_html`. These names aim to be consistent with `caption_markdown` and `caption_html`.

In v1, messages coming from a client used that client's parse mode as some sort of "global state". Based on the client's parse mode, v1 `message.text` property would return different things. But not *all* messages did this! Those coming from the raw API had no client, so `text` couldn't know how to format the message.

Overall, the old design made the parse mode be pretty hidden. This was not very intuitive and also made it very awkward to combine multiple parse modes.

The `forward` property is now `forward_info`. The `forward_to` method is now simply `forward()`. This makes it more consistent with the rest of message methods.

The `is_reply`, `reply_to_msg_id` and `reply_to` properties are now `replied_message_id`. The `get_reply_message` method is now `get_replied_message()`. This should make it clear that you are not getting a reply to the current message, but rather the message it replied to.

The `to_id`, `via_input_bot`, `action_entities`, `button_count` properties are also gone. Some were kept for backwards-compatibility, some were redundant.

The `click` method no longer exists in the message. Instead, find the right `buttons` to click on.

The `download` method no longer exists in the message. Instead, use `download` on the message's `file`.

HMMMM WEB\_PREVIEW VS LINK\_PREVIEW... probs use link. we're previewing a link not the web

## 4.2.10 Event and filters are now separate

Event types are no longer callable and do not have filters inside them. There is no longer nested class `Event` inside them either.

Instead, the event type itself is what the handler will actually be called with. Because filters are separate, there is no longer a need for `v1 @events.register` either. It also means you can combine filters with `&`, `|` and `~`.

Filters are now normal functions that work with any event. Of course, this doesn't mean all filters make sense for all events. But you can use them in a unified manner.

Filters no longer support asynchronous operations, which removes a footgun. This was most commonly experienced when using usernames as the `chats` filter in `v1`, and getting flood errors you couldn't handle. In `v2`, you must pass a list of identifiers. This means getting those identifiers is up to you, and you can handle it in a way that is appropriated for your application.

### See also:

In-depth explanation for [Updates](#).

## 4.2.11 Behaviour changes in events

Events produced by the client itself will now also be received as updates. This means, for example, that your `events.NewMessage` handlers will run when you use `Client.send_message()`. This is needed to properly handle updates.

In `v1`, there was a backwards-compatibility hack that flagged results from the client as their “own”. But in some rare cases, it was possible to still receive messages sent by the client itself in `v1`. The hack has been removed so now the library will consistently deliver all updates.

`events.StopPropagation` no longer exists. In `v1`, all handlers were always called. Now handlers are called in order until the filter for one returns `True`. The default behaviour is that handlers after that one are not called. This behaviour can be changed with the `check_all_handlers` flag in `Client` constructor.

`events.CallbackQuery` has been renamed to `events.ButtonCallback` and no longer also handles “inline bot callback queries”. This was a hacky workaround.

`events.MessageRead` no longer triggers when the `contents` of a message are read, such as voice notes being played.

Albums in Telegram are an illusion. There is no “album media”. There is only separate messages pretending to be a single message.

`events.Album` was a hack that waited for a small amount of time to group messages sharing the same grouped identifier. If you want to wait for a full album, you will need to wait yourself:

```
pending_albums = {} # global for simplicity
async def gather_album(event, handler):
    if pending := pending_albums.get(event.grouped_id):
        pending.append(event)
    else:
        pending_albums[event.grouped_id] = [event]
        # Wait for other events to come in. Adjust delay to your needs.
        # This will NOT work if sequential updates are enabled (spawn a task to do the
        ↪rest instead).
        await asyncio.sleep(1)
        events = pending_albums.pop(grouped_id, [])
        await handler(events)

@client.on(events.NewMessage)
```

(continues on next page)

(continued from previous page)

```

async def handler(event):
    if event.grouped_id:
        await gather_album(event, handle_album)
    else:
        await handle_message(event)

async def handle_album(events):
    ... # do stuff with events

async def handle_message(event):
    ... # do stuff with event

```

Note that the above code is not foolproof and will not handle more than one client. It might be possible for album events to be delayed for more than a second.

Note that messages that do **not** belong to an album can be received in-between an album.

Overall, it's probably better if you treat albums for what they really are: separate messages sharing a *grouped\_id*.

#### 4.2.12 Streamlined chat, input\_chat and chat\_id

The same goes for sender, input\_sender and sender\_id. And also for get\_chat, get\_input\_chat, get\_sender and get\_input\_sender. Yeah, it was bad.

Instead, events with chat information now *always* have a `.chat`, with *at least* the `.id`. The same is true for the `.sender`, as long as the event has one with at least the user identifier.

This doesn't mean the `.chat` or `.sender` will have all the information. Telegram may still choose to send their min version with only basic details. But it means you don't have to remember 5 different ways of using chats.

To replace the concept of "input chats", v2 introduces `types.PackedChat`. A "packed chat" is a chat with *just* enough information that you can use it without relying on Telethon's cache. This is the most efficient way to call methods like `Client.send_message()` too.

The concept of "marked IDs" also no longer exists. This means v2 no longer supports the `-` or `-100` prefixes on identifiers. `Peer`-wrapping is gone, too. Instead, you're strongly encouraged to use `types.PackedChat` instances.

The concepts of "entity" or "peer" are unified to simply `chat`. Overall, dealing with users, groups and channels should feel a lot more natural.

##### See also:

In-depth explanation for *Chats*.

Other methods like `client.get_peer_id`, `client.get_input_entity` and `client.get_entity` are gone too. While not directly related, `client.is_bot` is gone as well. You can use `Client.get_me()` or read it from the session instead.

The `telethon.utils` package is gone entirely, so methods like `utils.resolve_id` no longer exist either.

### 4.2.13 Session cache no longer exists

At least, not the way it did before.

The v1 cache that allowed you to use just chat identifiers to call methods is no longer saved to disk.

Sessions now only contain crucial information to have a working client. This includes the server address, authorization key, update state, and some very basic details.

To work around this, you can use `types.PackedChat`, which is designed to be easy to store. This means your application can choose the best way to deal with them rather than being forced into Telethon's session.

**See also:**

In-depth explanation for [Sessions](#).

### 4.2.14 StringSession no longer exists

If you need to serialize the session data to a string, you can use something like `jsonpickle`. Or even the built-in `pickle` followed by `base64` or just `bytes.hex()`. But be aware that these approaches probably will not be compatible with additions to the [Session](#).

## 4.3 Frequently Asked Questions

## 4.4 Contributing

Telethon welcomes all new contributions, whether it's reporting bugs or sending code patches.

Please keep both the philosophy and coding style below in mind.

Be mindful when adding new features. Every new feature must be understood by the maintainer, or otherwise it will probably rot. The *usefulness* : *maintenance-cost* ratio must be high enough to warrant being built-in. Consider whether your new features could be a separate add-on project entirely.

### 4.4.1 Philosophy

- Dependencies should only be added when absolutely necessary.
- Dependencies written in anything other than Python cannot be mandatory.
- The library must work correctly with no system dependencies other than Python 3.
- Strict type-checking is required to pass everywhere in the library to make upgrades easier.
- The code structure must make use of hard and clear boundaries to keep the different parts decoupled.
- The API should cover only the most commonly used features to avoid bloat and reduce maintenance costs.
- Documentation must be a pleasure to use and contain plenty of code examples.

## 4.4.2 Coding style

Knowledge of Python is obviously a must to develop a Python library. A good online resource is [Dive Into Python 3](#).

Telethon uses multiple tools to automatically format the code and check for linting rules. This means you can simply ignore formatting and let the tools handle it for you. You can find these tools under the `tools/` folder. See [tools/](#) below for an explanation.

The documentation is written with mostly a newline after every period. This is not a hard rule. Lines can be cut earlier if they become too long to be comfortable.

Commit messages should be short and descriptive. They should start with an action in the present (“Fix” and not “Fixed”). This saves a few characters and represents what the commit will “do” after applied.

## 4.4.3 Project structure

The repository contains several folders, each with their own “package”.

### **benches/**

This folder contains different benchmarks. Pretty straightforward.

### **stubs/**

If a dependency doesn’t support typing, files here must work around that.

### **tools/**

Various utility scripts. Each script should have a “comment” at the top explaining what they are for.

### **Code generation**

This will take `api.tl` and `mtproto.tl` files and generate `client/_impl/tl`.

```
pip install -e generator/  
python tools/codegen.py
```

### **Linting**

This includes format checks, type-checking and testing.

```
pip install -e client/[dev]  
python tools/check.py
```

## Documentation

Requires [sphinx](#) and [graphviz](#)'s `dot`.

```
pip install -e client/[doc]
python tools/docgen.py
```

Note that multiple optional dependency sets can be specified by separating them with a comma (`[dev, doc]`).

### generator/

A package that should not be published and is only used when developing the library. The implementation is private and exists under the `src/*/_impl/` folder. Only select parts are exported under public modules. Tests live under `tests/`.

The implementation consists of a parser and a code generator.

The parser is able to read parse `.tl` files (*Type Language* definition files). It doesn't do anything with the files other than to represent the content as Python objects.

---

### Type Language brief

TL-definitions are statements terminated with a semicolon `;` and often defined in a single line:

```
geoPointEmpty#1117dd5f = GeoPoint;
geoPoint#b2a2f663 flags:# long:double lat:double access_hash:long accuracy_radius:flags.
↳0?int = GeoPoint;
```

The first word is the name, optionally followed by the hash sign `#` and an hexadecimal number. Every definition can have a constructor identifier inferred based on its own text representation. The hexadecimal number will override the constructor identifier used for the definition.

What follows up to the equals-sign `=` are the fields of the definition. They have a name and a type, separated by the colon `:`.

The type `#` represents a bitflag. Other fields can be conditionally serialized by prefixing the type with `flag_name.bit_index?`.

After the equal-sign comes the name for the “base class”. This representation is known as “boxed”, and it contains the constructor identifier to discriminate a definition. If the definition name appears on its own, it will be “bare” and will not have the constructor identifier prefix.

---

The code generator uses the parsed definitions to generate Python code. Most of the code to serialize and deserialize objects lives under `serde/`.

An in-memory “filesystem” structure is kept before writing all files to disk. This makes it possible to execute most of the process in a sans-io manner. Once the code generation finishes, all files are written to disk at once.

See [tools/](#) above to learn how to generate code.



## client/

The Telethon client library and documentation lives here. This is the package that gets published. The implementation is private and exists under the `src/*/_impl/` folder. Only select parts are exported under public modules. Tests live under `tests/`.

The client implementation consists of several subpackages.

The `tl` package sits at the bottom. It is where the generated code is placed. It also contains some of the definitions needed for the generated code to work. Even though all the *RPC* live here, this package can't do anything by itself.

The `crypto` package implements all the encryption and decryption rules used by Telegram. Details concerning the *MTPProto* are mostly avoided, so the package can be generally useful.

The `mtproto` package implements the logic required to talk to Telegram. It is implemented in a sans-io manner. This package is responsible for generating an authorization key and serializing packets. It also contains some optimizations which are not strictly necessary when implementing the library.

The `mtsender` package simply adds IO to `mtproto`. It is responsible for driving the network, enqueueing requests, and waiting for results.

The `session` crate implements what's needed to manage the *session* state. The logic to handle and correctly order updates also lives here, in a sans-io manner.

The `client` ties everything together. This is what defines the Pythonic API to interact with Telegram. Custom object and event types also live here.

Even though only common methods are implemented, the code is still huge. For this reason, the *Client* implementation is separated from the class definition. The class definition only contains documentation and calls functions defined in other files. A tool under `tools/` exists to make it easy to keep these two in sync.

If you plan to port the library to a different language, good luck! You will need a code generator, the `crypto`, `mtproto` and `mtsender` packages to have an initial working version. The tests are your friend, write them too!



## PYTHON MODULE INDEX

### e

`telethon.events`, 60

`telethon.events.filters`, 63

### S

`telethon.session`, 91

### t

`telethon.types`, 67

`telethon.types.buttons`, 85



## A

access hash, **31**  
 access\_hash (*telethon.types.PackedChat* attribute), 81  
 add\_event\_handler() (*telethon.Client* method), 34  
 add\_photo() (*telethon.types.AlbumBuilder* method), 68  
 add\_video() (*telethon.types.AlbumBuilder* method), 68  
 admin\_rights (*telethon.types.Participant* property), 82  
 AdminRight (*class in telethon.types*), 67  
 AlbumBuilder (*class in telethon.types*), 68  
 All (*class in telethon.events.filters*), 63  
 answer() (*telethon.events.ButtonCallback* method), 60  
 Any (*class in telethon.events.filters*), 63  
 AsyncList (*class in telethon.types*), 69  
 AsyncReader (*class in telethon.\_impl.mtsender.sender*), 89  
 AsyncWriter (*class in telethon.\_impl.mtsender.sender*), 89  
 AUDIO (*telethon.events.filters.Media* attribute), 65  
 audio (*telethon.types.Message* property), 78  
 auth (*telethon.session.DataCenter* attribute), 93

## B

BAN\_USERS (*telethon.types.AdminRight* attribute), 67  
 banned (*telethon.types.Participant* property), 83  
 bot (*telethon.session.User* attribute), 94  
 BOT (*telethon.types.PackedType* attribute), 82  
 bot (*telethon.types.User* property), 84  
 Bot API, **31**  
 bot\_sign\_in() (*telethon.Client* method), 35  
 BROADCAST (*telethon.types.PackedType* attribute), 82  
 Button (*class in telethon.types*), 70  
 ButtonCallback (*class in telethon.events*), 60  
 buttons (*telethon.types.Message* property), 78

## C

Callback (*class in telethon.types.buttons*), 85  
 CallbackAnswer (*class in telethon.types*), 70  
 can\_forward (*telethon.types.Message* property), 78  
 CHANGE\_INFO (*telethon.types.AdminRight* attribute), 67  
 CHANGE\_INFO (*telethon.types.ChatRestriction* attribute), 72  
 Channel (*class in telethon.types*), 70

channel\_id (*telethon.events.MessageDeleted* property), 62  
 channels (*telethon.session.UpdateState* attribute), 94  
 ChannelState (*class in telethon.session*), 95  
 chat, **31**  
 Chat (*class in telethon.types*), 71  
 chat (*telethon.events.MessageRead* property), 62  
 chat (*telethon.types.Dialog* property), 73  
 chat (*telethon.types.Draft* property), 74  
 chat (*telethon.types.Message* property), 78  
 CHAT (*telethon.types.PackedType* attribute), 82  
 chat\_ids (*telethon.events.filters.Chats* property), 64  
 ChatRestriction (*class in telethon.types*), 72  
 Chats (*class in telethon.events.filters*), 64  
 ChatType (*class in telethon.events.filters*), 64  
 check\_password() (*telethon.Client* method), 35  
 click() (*telethon.types.buttons.Callback* method), 85  
 click() (*telethon.types.buttons.Text* method), 86  
 Client (*class in telethon*), 33  
 client (*telethon.events.Event* property), 61  
 close() (*telethon.\_impl.mtsender.sender.AsyncWriter* method), 89  
 code (*telethon.RpcError* property), 87  
 Combinable (*class in telethon.\_impl.client.events.filters.combinators*), 88  
 Command (*class in telethon.events.filters*), 64  
 connect() (*telethon.Client* method), 36  
 connected (*telethon.Client* property), 36  
 Connector (*class in telethon.\_impl.mtsender.sender*), 90  
 Continue (*class in telethon.events*), 61  
 creator (*telethon.types.Participant* property), 83

## D

Data (*class in telethon.events.filters*), 65  
 data (*telethon.events.ButtonCallback* property), 60  
 data (*telethon.types.buttons.Callback* property), 85  
 DataCenter (*class in telethon.session*), 93  
 date (*telethon.session.UpdateState* attribute), 94  
 date (*telethon.types.Draft* property), 74  
 date (*telethon.types.Message* property), 78  
 dc (*telethon.session.User* attribute), 94  
 dcs (*telethon.session.Session* attribute), 93

- `delete()` (*telethon.session.MemorySession* method), 92
  - `delete()` (*telethon.session.SqliteSession* method), 91
  - `delete()` (*telethon.session.Storage* method), 91
  - `delete()` (*telethon.types.Draft* method), 74
  - `delete()` (*telethon.types.Message* method), 78
  - `delete_dialog()` (*telethon.Client* method), 36
  - `DELETE_MESSAGES` (*telethon.types.AdminRight* attribute), 67
  - `delete_messages()` (*telethon.Client* method), 37
  - `DELETE_STORIES` (*telethon.types.AdminRight* attribute), 67
  - `description` (*telethon.types.InlineResult* property), 77
  - `Dialog` (class in *telethon.types*), 73
  - `disconnect()` (*telethon.Client* method), 37
  - `download()` (*telethon.Client* method), 38
  - `download()` (*telethon.types.File* method), 75
  - `Draft` (class in *telethon.types*), 73
  - `draft` (*telethon.types.Dialog* property), 73
  - `drain()` (*telethon.\_impl.mtsender.sender.AsyncWriter* method), 89
- ## E
- `edit()` (*telethon.types.Draft* method), 74
  - `edit()` (*telethon.types.Message* method), 78
  - `edit_draft()` (*telethon.Client* method), 38
  - `edit_message()` (*telethon.Client* method), 39
  - `EDIT_MESSAGES` (*telethon.types.AdminRight* attribute), 67
  - `EDIT_STORIES` (*telethon.types.AdminRight* attribute), 67
  - `EMBED_LINKS` (*telethon.types.ChatRestriction* attribute), 72
  - `errors` (in module *telethon*), 87
  - `Event` (class in *telethon.events*), 61
  - `ext` (*telethon.types.File* property), 75
- ## F
- `File` (class in *telethon.types*), 75
  - `file` (*telethon.types.Message* property), 79
  - `filter` (*telethon.events.filters*.Not property), 66
  - `filters` (*telethon.events.filters*.All property), 63
  - `filters` (*telethon.events.filters*.Any property), 64
  - `first_name` (*telethon.types.User* property), 84
  - `Forward` (class in *telethon.events.filters*), 65
  - `forward()` (*telethon.types.Message* method), 79
  - `forward_info` (*telethon.types.Message* property), 79
  - `forward_messages()` (*telethon.Client* method), 40
  - `from_bytes()` (*telethon.types.PackedChat* class method), 81
  - `from_hex()` (*telethon.types.PackedChat* class method), 82
- ## G
- `get_admin_log()` (*telethon.Client* method), 40
  - `get_chats()` (*telethon.Client* method), 41
  - `get_contacts()` (*telethon.Client* method), 41
  - `get_dialogs()` (*telethon.Client* method), 42
  - `get_drafts()` (*telethon.Client* method), 42
  - `get_file_bytes()` (*telethon.Client* method), 42
  - `get_handler_filter()` (*telethon.Client* method), 43
  - `get_me()` (*telethon.Client* method), 43
  - `get_message()` (*telethon.events.ButtonCallback* method), 60
  - `get_messages()` (*telethon.Client* method), 44
  - `get_messages_with_ids()` (*telethon.Client* method), 44
  - `get_participants()` (*telethon.Client* method), 45
  - `get_profile_photos()` (*telethon.Client* method), 45
  - `get_replied_message()` (*telethon.types.Message* method), 79
  - `GIGAGROUP` (*telethon.types.PackedType* attribute), 82
  - `Group` (class in *telethon.types*), 75
  - `grouped_id` (*telethon.types.Message* property), 79
- ## H
- `height` (*telethon.types.File* property), 75
  - `hex` (*telethon.types.PackedChat* property), 82
  - `hint` (*telethon.types.PasswordToken* property), 83
  - `HTTP Bot API`, 31
- ## I
- `id` (*telethon.session.ChannelState* attribute), 95
  - `id` (*telethon.session.DataCenter* attribute), 93
  - `id` (*telethon.session.User* attribute), 94
  - `id` (*telethon.types.Channel* property), 70
  - `id` (*telethon.types.Chat* property), 71
  - `id` (*telethon.types.Group* property), 76
  - `id` (*telethon.types.Message* property), 79
  - `id` (*telethon.types.PackedChat* attribute), 82
  - `id` (*telethon.types.RecentAction* property), 84
  - `id` (*telethon.types.User* property), 84
  - `Incoming` (class in *telethon.events.filters*), 65
  - `incoming` (*telethon.types.Message* property), 79
  - `InFileLike` (class in *telethon.\_impl.client.types.file*), 88
  - `inline_query()` (*telethon.Client* method), 45
  - `InlineButton` (class in *telethon.types*), 76
  - `InlineQuery` (class in *telethon.events*), 61
  - `InlineResult` (class in *telethon.types*), 77
  - `interactive_login()` (*telethon.Client* method), 46
  - `INVITE_USERS` (*telethon.types.AdminRight* attribute), 67
  - `INVITE_USERS` (*telethon.types.ChatRestriction* attribute), 72
  - `ipv4_addr` (*telethon.session.DataCenter* attribute), 93
  - `ipv6_addr` (*telethon.session.DataCenter* attribute), 94
  - `is_authorized()` (*telethon.Client* method), 47
  - `is_channel()` (*telethon.types.PackedChat* method), 82
  - `is_chat()` (*telethon.types.PackedChat* method), 82
  - `is_megagroup` (*telethon.types.Group* property), 76
  - `is_user()` (*telethon.types.PackedChat* method), 82

## L

last\_name (*telethon.types.User* property), 84  
 latest\_message (*telethon.types.Dialog* property), 73  
 layer, 31  
 left (*telethon.types.Participant* property), 83  
 link\_preview (*telethon.types.Draft* property), 74  
 link\_preview (*telethon.types.Message* property), 79  
 load() (*telethon.session.MemorySession* method), 92  
 load() (*telethon.session.SqliteSession* method), 91  
 load() (*telethon.session.Storage* method), 91  
 login, 31  
 LoginToken (*class in telethon.types*), 77

## M

MANAGE\_ADMINS (*telethon.types.AdminRight* attribute), 67  
 MANAGE\_CALLS (*telethon.types.AdminRight* attribute), 67  
 MANAGE\_TOPICS (*telethon.types.AdminRight* attribute), 67  
 MANAGE\_TOPICS (*telethon.types.ChatRestriction* attribute), 72  
 max\_message\_id\_read (*telethon.events.MessageRead* property), 62  
 Media (*class in telethon.events.filters*), 65  
 MEGAGROUP (*telethon.types.PackedType* attribute), 82  
 MemorySession (*class in telethon.session*), 92  
 Message (*class in telethon.types*), 77  
 message\_ids (*telethon.events.MessageDeleted* property), 62  
 MessageDeleted (*class in telethon.events*), 61  
 MessageEdited (*class in telethon.events*), 62  
 MessageRead (*class in telethon.events*), 62  
 module  
   *telethon.events*, 60  
   *telethon.events.filters*, 63  
   *telethon.session*, 90  
   *telethon.types*, 67  
   *telethon.types.buttons*, 85  
 MTPProto, 31

## N

name (*telethon.RpcError* property), 87  
 name (*telethon.types.Channel* property), 71  
 name (*telethon.types.Chat* property), 71  
 name (*telethon.types.File* property), 75  
 name (*telethon.types.Group* property), 76  
 name (*telethon.types.User* property), 84  
 NewMessage (*class in telethon.events*), 62  
 Not (*class in telethon.events.filters*), 65

## O

on() (*telethon.Client* method), 47  
 OTHER (*telethon.types.AdminRight* attribute), 67

OutFileLike (*class in telethon.\_impl.client.types.file*), 89  
 Outgoing (*class in telethon.events.filters*), 66  
 outgoing (*telethon.types.Message* property), 79

## P

pack() (*telethon.types.Channel* method), 71  
 pack() (*telethon.types.Chat* method), 71  
 pack() (*telethon.types.Group* method), 76  
 pack() (*telethon.types.User* method), 84  
 PackedChat (*class in telethon.types*), 81  
 PackedType (*class in telethon.types*), 82  
 Participant (*class in telethon.types*), 82  
 PasswordToken (*class in telethon.types*), 83  
 phone (*telethon.types.User* property), 84  
 PHOTO (*telethon.events.filters.Media* attribute), 65  
 photo (*telethon.types.Message* property), 80  
 pin() (*telethon.types.Message* method), 80  
 pin\_message() (*telethon.Client* method), 48  
 PIN\_MESSAGES (*telethon.types.AdminRight* attribute), 67  
 PIN\_MESSAGES (*telethon.types.ChatRestriction* attribute), 72  
 POST\_MESSAGES (*telethon.types.AdminRight* attribute), 68  
 POST\_STORIES (*telethon.types.AdminRight* attribute), 68  
 prepare\_album() (*telethon.Client* method), 48  
 pts (*telethon.session.ChannelState* attribute), 95  
 pts (*telethon.session.UpdateState* attribute), 94

## Q

qts (*telethon.session.UpdateState* attribute), 94  
 query (*telethon.types.buttons.SwitchInline* property), 86

## R

Raw API, 31  
 read() (*telethon.\_impl.client.types.file.InFileLike* method), 89  
 read() (*telethon.\_impl.mtsender.sender.AsyncReader* method), 89  
 read() (*telethon.types.Message* method), 80  
 read\_message() (*telethon.Client* method), 48  
 RecentAction (*class in telethon.types*), 83  
 REMAIN\_ANONYMOUS (*telethon.types.AdminRight* attribute), 68  
 remove\_event\_handler() (*telethon.Client* method), 49  
 replied\_message\_id (*telethon.types.Draft* property), 74  
 replied\_message\_id (*telethon.types.Message* property), 80  
 Reply (*class in telethon.events.filters*), 66  
 reply() (*telethon.types.Message* method), 80  
 request\_login\_code() (*telethon.Client* method), 49  
 RequestGeoLocation (*class in telethon.types.buttons*), 85

RequestPhone (class in telethon.types.buttons), 85  
 RequestPoll (class in telethon.types.buttons), 85  
 resolve\_username() (telethon.Client method), 50  
 respond() (telethon.types.Message method), 80  
 restrictions (telethon.types.Participant property), 83  
 RPC, 31  
 RPC Error, 31  
 RpcError (class in telethon), 87  
 run\_until\_disconnected() (telethon.Client method), 50

## S

save() (telethon.session.MemorySession method), 92  
 save() (telethon.session.SqliteSession method), 92  
 save() (telethon.session.Storage method), 91  
 search\_all\_messages() (telethon.Client method), 50  
 search\_messages() (telethon.Client method), 51  
 send() (telethon.types.AlbumBuilder method), 69  
 send() (telethon.types.Draft method), 74  
 send() (telethon.types.InlineResult method), 77  
 send\_audio() (telethon.Client method), 51  
 SEND\_AUDIOS (telethon.types.ChatRestriction attribute), 72  
 SEND\_DOCUMENTS (telethon.types.ChatRestriction attribute), 72  
 send\_file() (telethon.Client method), 52  
 SEND\_GAMES (telethon.types.ChatRestriction attribute), 72  
 SEND\_GIFS (telethon.types.ChatRestriction attribute), 72  
 SEND\_INLINE (telethon.types.ChatRestriction attribute), 72  
 SEND\_MEDIA (telethon.types.ChatRestriction attribute), 72  
 send\_message() (telethon.Client method), 54  
 SEND\_MESSAGES (telethon.types.ChatRestriction attribute), 72  
 send\_photo() (telethon.Client method), 55  
 SEND\_PHOTOS (telethon.types.ChatRestriction attribute), 72  
 SEND\_PLAIN\_MESSAGES (telethon.types.ChatRestriction attribute), 73  
 SEND\_POLLS (telethon.types.ChatRestriction attribute), 73  
 SEND\_ROUND\_VIDEOS (telethon.types.ChatRestriction attribute), 73  
 SEND\_STICKERS (telethon.types.ChatRestriction attribute), 73  
 send\_video() (telethon.Client method), 55  
 SEND\_VIDEOS (telethon.types.ChatRestriction attribute), 73  
 SEND\_VOICE\_NOTES (telethon.types.ChatRestriction attribute), 73  
 sender (telethon.types.Message property), 80  
 sender\_ids (telethon.events.filters.Senders property), 66

Senders (class in telethon.events.filters), 66  
 seq (telethon.session.UpdateState attribute), 94  
 session, 31  
 Session (class in telethon.session), 93  
 session (telethon.session.MemorySession attribute), 92  
 set\_admin\_rights() (telethon.types.Participant method), 83  
 set\_chat\_default\_restrictions() (telethon.Client method), 56  
 set\_default\_restrictions() (telethon.types.Group method), 76  
 set\_handler\_filter() (telethon.Client method), 57  
 set\_participant\_admin\_rights() (telethon.Client method), 57  
 set\_participant\_restrictions() (telethon.Client method), 58  
 set\_restrictions() (telethon.types.Participant method), 83  
 sign in, 31  
 sign\_in() (telethon.Client method), 59  
 sign\_out() (telethon.Client method), 59  
 silent (telethon.types.Message property), 81  
 SqliteSession (class in telethon.session), 91  
 state (telethon.session.Session attribute), 93  
 Storage (class in telethon.session), 91  
 SwitchInline (class in telethon.types.buttons), 86

## T

T (in module telethon.\_impl.client.types.async\_list), 88  
 telethon.events  
   module, 60  
 telethon.events.filters  
   module, 63  
 telethon.session  
   module, 90  
 telethon.types  
   module, 67  
 telethon.types.buttons  
   module, 85  
 Text (class in telethon.events.filters), 66  
 Text (class in telethon.types.buttons), 86  
 text (telethon.types.Button property), 70  
 text (telethon.types.buttons.Text property), 86  
 text (telethon.types.CallbackAnswer property), 70  
 text (telethon.types.Draft property), 75  
 text (telethon.types.Message property), 81  
 text\_html (telethon.types.Draft property), 75  
 text\_html (telethon.types.Message property), 81  
 text\_markdown (telethon.types.Draft property), 75  
 text\_markdown (telethon.types.Message property), 81  
 thumbnails (telethon.types.File property), 75  
 timeout (telethon.types.LoginToken property), 77  
 title (telethon.types.InlineResult property), 77  
 TL, 32



ty (*telethon.types.PackedChat* attribute), 82  
 type (*telethon.events.filters.ChatType* property), 64  
 type (*telethon.types.InlineResult* property), 77  
 Type Language, 32  
 types (*telethon.events.filters.Media* property), 65

## U

unpin() (*telethon.types.Message* method), 81  
 unpin\_message() (*telethon.Client* method), 59  
 unread\_count (*telethon.types.Dialog* property), 73  
 UpdateState (class in *telethon.session*), 94  
 Url (class in *telethon.types.buttons*), 86  
 url (*telethon.types.buttons.Url* property), 86  
 url (*telethon.types.CallbackAnswer* property), 70  
 User (class in *telethon.session*), 94  
 User (class in *telethon.types*), 84  
 user (*telethon.session.Session* attribute), 93  
 USER (*telethon.types.PackedType* attribute), 82  
 user (*telethon.types.Participant* property), 83  
 username (*telethon.session.User* attribute), 94  
 username (*telethon.types.Channel* property), 71  
 username (*telethon.types.Chat* property), 72  
 username (*telethon.types.Group* property), 76  
 username (*telethon.types.User* property), 84

## V

value (*telethon.RpcError* property), 87  
 VERSION (*telethon.session.Session* attribute), 93  
 VIDEO (*telethon.events.filters.Media* attribute), 65  
 video (*telethon.types.Message* property), 81  
 VIEW\_MESSAGES (*telethon.types.ChatRestriction* attribute), 73

## W

wait\_closed() (*telethon.\_impl.mtsender.sender.AsyncWriter* method), 89  
 width (*telethon.types.File* property), 75  
 write() (*telethon.\_impl.client.types.file.OutFileLike* method), 89  
 write() (*telethon.\_impl.mtsender.sender.AsyncWriter* method), 90

## Y

yourself, 32