Telethon Documentation

Release 1.9.0

Lonami

Jul 06, 2019

First Steps

```python
from telethon.sync import TelegramClient, events

with TelegramClient('name', api_id, api_hash) as client:
    client.send_message('me', 'Hello, myself!')
    print(client.download_profile_photo('me'))

    @client.on(events.NewMessage(pattern='(?i).*Hello'))
    async def handler(event):
        await event.reply('Hey!')

    client.run_until_disconnected()
```

- Are you new here? Jump straight into *Installation*!

- Looking for the method reference? See *Client Reference*.

- Did you upgrade the library? Please read *Changelog (Version History)*.

- Used Telethon before v1.0? See *Compatibility and Convenience*.

- Coming from Bot API or want to create new bots? See *HTTP Bot API vs MTProto*.

- Need the full API reference? https://tl.telethon.dev/.

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 heavy job for you, so you can focus on developing an application.

How should I use the documentation?

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.

2.1 Installation

Telethon is a Python library, which means you need to download and install Python from https://www.python.org/downloads/ if you haven't already. Once you have Python installed, run:

```
pip3 install -U telethon --user
```

To install or upgrade the library to the latest version.

2.1.1 Installing Development Versions

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

```
pip3 install -U https://github.com/LonamiWebs/Telethon/archive/master.zip --user
```

Note: The development version may have bugs and is not recommended for production use. However, when you are reporting a library bug, you should try if the bug still occurs in this version.

2.1.2 Verification

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

```
python3 -c "import telethon; print(telethon.__version__)"
```

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

2.1.3 Optional Dependencies

If cryptg is installed, **the library will work a lot faster**, since encryption and decryption will be made in C instead of Python. If your code deals with a lot of updates or you are downloading/uploading a lot of files, you will notice a considerable speed-up (from a hundred kilobytes per second to several megabytes per second, if your connection allows it). If it's not installed, pyaes will be used (which is pure Python, so it's much slower).

If pillow is installed, large images will be automatically resized when sending photos to prevent Telegram from failing with "invalid image". Official clients also do this.

If aiohttp is installed, the library will be able to download WebDocument media files (otherwise you will get an error).

If hachoir is installed, it will be used to extract metadata from files when sending documents. Telegram uses this information to show the song's performer, artist, title, duration, and for videos too (including size). Otherwise, they will default to empty values, and you can set the attributes manually.

Note: Some of the modules may require additional dependencies before being installed through `pip`. If you have an `apt`-based system, consider installing the most commonly missing dependencies:

```
apt update
apt install clang lib{jpeg-turbo,webp}-dev python{,-dev} zlib-dev
pip install -U --user setuptools
pip install -U --user telethon cryptg pillow
```

Thanks to @bb010g for writing down this nice list.

2.2 Signing In

Before working with Telegram's API, you need to get your own 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!

Note: This API ID and hash is the one used by *your application*, not your phone number. You can use this API ID and hash with *any* phone number or even for bot accounts.

2.2.1 Editing the Code

This is a little introduction for those new to Python programming in general.

We will write our code inside `hello.py`, so you can use any text editor that you like. To run the code, use `python3 hello.py` from the terminal.

Important: Don't call your script `telethon.py`! Python will try to import the client from there and it will fail with an error such as "ImportError: cannot import name 'TelegramClient' …".

2.2.2 Signing In

We can finally write some code to log into our account!

```python
from telethon.sync import TelegramClient

# Use your own values from my.telegram.org
api_id = 12345
api_hash = '0123456789abcdef0123456789abcdef'

# The first parameter is the .session file name (absolute paths allowed)
with TelegramClient('anon', api_id, api_hash) as client:
    client.send_message('me', 'Hello, myself!')
```

In the first line, we import the class name so we can create an instance of the client. Then, we define variables to store our API ID and hash conveniently.

At last, we create a new *TelegramClient* instance and call it `client`. We can now use the client variable for anything that we want, such as sending a message to ourselves.

Using a `with` block is the preferred way to use the library. It will automatically *start()* the client, logging or signing up if necessary.

If the `.session` file already existed, it will not login again, so be aware of this if you move or rename the file!

2.2.3 Signing In as a Bot Account

You can also use Telethon for your bots (normal bot accounts, not users). You will still need an API ID and hash, but the process is very similar:

```python
from telethon.sync import TelegramClient

api_id = 12345
api_hash = '0123456789abcdef0123456789abcdef'
bot_token = '12345:0123456789abcdef0123456789abcdef

# We have to manually call "start" if we want an explicit bot token
bot = TelegramClient('bot', api_id, api_hash).start(bot_token=bot_token)

# But then we can use the client instance as usual
with bot:
    ...
```

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

2.2.4 Signing In behind a Proxy

If you need to use a proxy to access Telegram, you will need to install PySocks and then change:

```
TelegramClient('anon', api_id, api_hash)
```

with

```
TelegramClient('anon', api_id, api_hash, proxy=(socks.SOCKS5, '127.0.0.1', 4444))
```

(of course, replacing the IP and port with the IP and port of the proxy).

The `proxy=` argument should be a tuple, a list or a dict, consisting of parameters described in PySocks usage.

2.2.5 Using MTProto Proxies

MTProto Proxies are Telegram's alternative to normal proxies, and work a bit differently. The following protocols are available:

- `ConnectionTcpMTProxyAbridged`
- `ConnectionTcpMTProxyIntermediate`
- `ConnectionTcpMTProxyRandomizedIntermediate` (preferred)

For now, you need to manually specify these special connection modes if you want to use a MTProto Proxy. Your code would look like this:

```python
from telethon import TelegramClient, connection
#   we need to change the connection ^^^^^^^^^^

client = TelegramClient(
    'anon',
    api_id,
    api_hash,

    # Use one of the available connection modes.
    # Normally, this one works with most proxies.
    connection=connection.ConnectionTcpMTProxyRandomizedIntermediate,

    # Then, pass the proxy details as a tuple:
    #     (host name, port, proxy secret)
    #
    # If the proxy has no secret, the secret must be:
    #     '00000000000000000000000000000000'
    proxy=('mtproxy.example.com', 2002, 'secret')
)
```

In future updates, we may make it easier to use MTProto Proxies (such as avoiding the need to manually pass `connection=`).

In short, the same code above but without comments to make it clearer:

```python
from telethon import TelegramClient, connection

client = TelegramClient(
    'anon', api_id, api_hash,
    connection=connection.ConnectionTcpMTProxyRandomizedIntermediate,
    proxy=('mtproxy.example.com', 2002, 'secret')
)
```

2.3 Quick-Start

Let's see a longer example to learn some of the methods that the library has to offer. These are known as "friendly methods", and you should always use these if possible.

```python
from telethon.sync import TelegramClient

# Remember to use your own values from my.telegram.org!
api_id = 12345
api_hash = '0123456789abcdef0123456789abcdef'

with TelegramClient('anon', api_id, api_hash) as client:
    # Getting information about yourself
    me = client.get_me()

    # "me" is an User object. You can pretty-print
    # any Telegram object with the "stringify" method:
    print(me.stringify())

    # When you print something, you see a representation of it.
    # You can access all attributes of Telegram objects with
    # the dot operator. For example, to get the username:
    username = me.username
    print(username)
    print(me.phone)

    # You can print all the dialogs/conversations that you are part of:
    for dialog in client.iter_dialogs():
        print(dialog.name, 'has ID', dialog.id)

    # You can send messages to yourself...
    client.send_message('me', 'Hello, myself!')
    # ...to some chat ID
    client.send_message(-100123456, 'Hello, group!')
    # ...to your contacts
    client.send_message('+34600123123', 'Hello, friend!')
    # ...or even to any username
    client.send_message('TelethonChat', 'Hello, Telethon!')

    # You can, of course, use markdown in your messages:
    message = client.send_message(
        'me',
        'This message has **bold**, `code`, __italics__ and '
        'a [nice website](https://lonamiwebs.github.io)!',
        link_preview=False
    )

    # Sending a message returns the sent message object, which you can use
    print(message.raw_text)

    # You can reply to messages directly if you have a message object
    message.reply('Cool!')

    # Or send files, songs, documents, albums...
    client.send_file('me', '/home/me/Pictures/holidays.jpg')

    # You can print the message history of any chat:
```

(continues on next page)

```
    for message in client.iter_messages('me'):
        print(message.id, message.text)

        # You can download media from messages, too!
        # The method will return the path where the file was saved.
        if message.photo:
            path = message.download_media()
            print('File saved to', path)
```

Here, we show how to sign in, get information about yourself, send messages, files, getting chats, printing messages, and downloading files.

You should make sure that you understand what the code shown here does, take note on how methods are called and used and so on before proceeding. We will see all the available methods later on.

2.4 Updates

Updates are an important topic in a messaging platform like Telegram. After all, you want to be notified when a new message arrives, when a member joins, when someone starts typing, etc. For that, you can use **events**.

Important: It is strongly advised to enable logging when working with events, since exceptions in event handlers are hidden by default. Please add the following snippet to the very top of your file:

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

2.4.1 Getting Started

Let's start things with an example to automate replies:

```
from telethon import TelegramClient, events

client = TelegramClient('anon', api_id, api_hash)

@client.on(events.NewMessage)
async def my_event_handler(event):
    if 'hello' in event.raw_text:
        await event.reply('hi!')

client.start()
client.run_until_disconnected()
```

This code isn't much, but there might be some things unclear. Let's break it down:

```
from telethon import TelegramClient, events

client = TelegramClient('anon', api_id, api_hash)
```

This is normal creation (of course, pass session name, API ID and hash). Nothing we don't know already.

```
@client.on(events.NewMessage)
```

This Python decorator will attach itself to the my_event_handler definition, and basically means that *on* a *NewMessage* event, the callback function you're about to define will be called:

```python
async def my_event_handler(event):
    if 'hello' in event.raw_text:
        await event.reply('hi!')
```

If a *NewMessage* event occurs, and 'hello' is in the text of the message, we *reply()* to the event with a 'hi!' message.

Note: Event handlers **must** be async def. After all, Telethon is an asynchronous library based on asyncio, which is a safer and often faster approach to threads.

You **must** await all method calls that use network requests, which is most of them.

2.4.2 More Examples

Replying to messages with hello is fun, but, can we do more?

```python
@client.on(events.NewMessage(outgoing=True, pattern=r'\.save'))
async def handler(event):
    if event.is_reply:
        replied = await event.get_reply_message()
        sender = replied.sender
        await client.download_profile_photo(sender)
        await event.respond('Saved your photo {}'.format(sender.username))
```

We could also get replies. This event filters outgoing messages (only those that we send will trigger the method), then we filter by the regex r'\.save', which will match messages starting with ".save".

Inside the method, we check whether the event is replying to another message or not. If it is, we get the reply message and the sender of that message, and download their profile photo.

Let's delete messages which contain "heck". We don't allow swearing here.

```python
@client.on(events.NewMessage(pattern=r'(?i).*heck'))
async def handler(event):
    await event.delete()
```

With the r'(?i).*heck' regex, we match case-insensitive "heck" anywhere in the message. Regex is very powerful and you can learn more at https://regexone.com/.

So far, we have only seen the *NewMessage*, but there are many more which will be covered later. This is only a small introduction to updates.

2.4.3 Entities

When you need the user or chat where an event occurred, you **must** use the following methods:

```python
async def handler(event):
    # Good
```

(continues on next page)

```
    chat = await event.get_chat()
    sender = await event.get_sender()
    chat_id = event.chat_id
    sender_id = event.sender_id

    # BAD. Don't do this
    chat = event.chat
    sender = event.sender
    chat_id = event.chat.id
    sender_id = event.sender.id
```

Events are like messages, but don't have all the information a message has! When you manually get a message, it will have all the information it needs. When you receive an update about a message, it **won't** have all the information, so you have to **use the methods**, not the properties.

Make sure you understand the code seen here before continuing! As a rule of thumb, remember that new message events behave just like message objects, so you can do with them everything you can do with a message object.

2.5 Next Steps

These basic first steps should have gotten you started with the library.

By now, you should know how to call friendly methods and how to work with the returned objects, how things work inside event handlers, etc.

Next, we will see a quick reference summary of *all* the methods and properties that you will need when using the library. If you follow the links there, you will expand the documentation for the method and property, with more examples on how to use them.

Therefore, **you can find an example on every method** of the client to learn how to use it, as well as a description of all the arguments.

After that, we will go in-depth with some other important concepts that are worth learning and understanding.

From now on, you can keep pressing the "Next" button if you want, or use the menu on the left, since some pages are quite lengthy.

2.6 FAQ

Let's start the quick references section with some useful tips to keep in mind, with the hope that you will understand why certain things work the way that they do.

Contents

- *FAQ*
    - *Code without errors doesn't work*
    - *How can I except FloodWaitError?*
    - *My account was deleted/limited when using the library*
    - *How can I use a proxy?*

- *How do I access a field?*
- *AttributeError: 'coroutine' object has no attribute 'id'*
- *sqlite3.OperationalError: database is locked*
- *event.chat or event.sender is None*
- *What does "bases ChatGetter" mean?*
- *Can I use Flask with the library?*
- *Can I use Anaconda/Spyder/IPython with the library?*

2.6.1 Code without errors doesn't work

Then it probably has errors, but you haven't enabled logging yet. To enable logging, at the following code to the top of your main file:

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

You can change the logging level to be something different, from less to more information:

```python
level=logging.CRITICAL   # won't show errors (same as disabled)
level=logging.ERROR      # will only show errors that you didn't handle
level=logging.WARNING    # will also show messages with medium severity, such as
→internal Telegram issues
level=logging.INFO       # will also show informational messages, such as connection
→or disconnections
level=logging.DEBUG      # will show a lot of output to help debugging issues in the
→library
```

See the official Python documentation for more information on logging.

2.6.2 How can I except FloodWaitError?

You can use all errors from the API by importing:

```python
from telethon import errors
```

And except them as such:

```python
try:
    client.send_message(chat, 'Hi')
except errors.FloodWaitError as e:
    # e.seconds is how many seconds you have
    # to wait before making the request again.
    print('Flood for', e.seconds)
```

2.6.3 My account was deleted/limited when using the library

The library will only do things that you tell it to do. If you use the library with bad intentions, Telegram will hopefully ban you.

However, you may also be part of a limited country, such as Iran or Russia. In that case, we have bad news for you. Telegram is much more likely to ban these numbers, as they are often used to spam other accounts, likely through the use of libraries like this one. The best advice we can give you is to not abuse the API, like calling many requests really quickly, and to sign up with these phones through an official application.

We have also had reports from Kazakhstan and China, where connecting would fail. To solve these connection problems, you should use a proxy.

Telegram may also ban virtual (VoIP) phone numbers, as again, they're likely to be used for spam.

If you want to check if your account has been limited, simply send a private message to @SpamBot through Telegram itself. You should notice this by getting errors like `PeerFloodError`, which means you're limited, for instance, when sending a message to some accounts but not others.

For more discussion, please see issue 297.

2.6.4 How can I use a proxy?

This was one of the first things described in *Signing In*.

2.6.5 How do I access a field?

This is basic Python knowledge. You should use the dot operator:

```python
me = client.get_me()
print(me.username)
#        ^ we used the dot operator to access the username attribute

result = client(functions.photos.GetUserPhotosRequest(
    user_id='me',
    offset=0,
    max_id=0,
    limit=100
))

# Working with list is also pretty basic
print(result.photos[0].sizes[-1].type)
#            ^        ^ ^      ^ ^
#            |        | |      | \ type
#            |        | |       \ last size
#            |        | \ list of sizes
#  access    |        \ first photo from the list
#  the...    \ list of photos
#
# To print all, you could do (or mix-and-match):
for photo in result.photos:
    for size in photo.sizes:
        print(size.type)
```

2.6.6 AttributeError: 'coroutine' object has no attribute 'id'

You either forgot to:

```python
import telethon.sync
#              ^^^^^ import sync
```

Or:

```python
async def handler(event):
    me = await client.get_me()
    #      ^^^^^ note the await
    print(me.username)
```

2.6.7 sqlite3.OperationalError: database is locked

An older process is still running and is using the same `'session'` file.

This error occurs when **two or more clients use the same session**, that is, when you write the same session name to be used in the client:

- You have an older process using the same session file.

- You have two different scripts running (interactive sessions count too).

- You have two clients in the same script running at the same time.

The solution is, if you need two clients, use two sessions. If the problem persists and you're on Linux, you can use `fuser my.session` to find out the process locking the file. As a last resort, you can reboot your system.

If you really dislike SQLite, use a different session storage. There is an entire section covering that at *Session Files*.

2.6.8 event.chat or event.sender is None

Telegram doesn't always send this information in order to save bandwidth. If you need the information, you should fetch it yourself, since the library won't do unnecessary work unless you need to:

```python
async def handler(event):
    chat = await event.get_chat()
    sender = await event.get_sender()
```

2.6.9 What does "bases ChatGetter" mean?

In Python, classes can base others. This is called inheritance. What it means is that "if a class bases another, you can use the other's methods too".

For example, *Message bases ChatGetter*. In turn, *ChatGetter* defines things like *obj.chat_id*.

So if you have a message, you can access that too:

```python
# ChatGetter has a chat_id property, and Message bases ChatGetter.
# Thus you can use ChatGetter properties and methods from Message
print(message.chat_id)
```

Telegram has a lot to offer, and inheritance helps the library reduce boilerplate, so it's important to know this concept. For newcomers, this may be a problem, so we explain what it means here in the FAQ.

2.6.10 Can I use Flask with the library?

Yes, if you know what you are doing. However, you will probably have a lot of headaches to get threads and asyncio to work together. Instead, consider using Quart, an asyncio-based alternative to Flask.

Check out quart_login.py for an example web-application based on Quart.

2.6.11 Can I use Anaconda/Spyder/IPython with the library?

Yes, but these interpreters run the asyncio event loop implicitly, which interferes with the `telethon.sync` magic module.

If you use them, you should **not** import `sync`:

```
# Change any of these...:
from telethon import TelegramClient, sync, ...
from telethon.sync import TelegramClient, ...

# ...with this:
from telethon import TelegramClient, ...
```

You are also more likely to get "sqlite3.OperationalError: database is locked" with them. If they cause too much trouble, just write your code in a `.py` file and run that, or use the normal `python` interpreter.

2.7 Client Reference

This page contains a summary of all the important methods and properties that you may need when using Telethon. They are sorted by relevance and are not in alphabetical order.

You should use this page to learn about which methods are available, and if you need an usage example or further description of the arguments, be sure to follow the links.

Contents

- *Client Reference*
  - *TelegramClient*
    * *Auth*
    * *Base*
    * *Messages*
    * *Uploads*
    * *Downloads*
    * *Dialogs*
    * *Users*
    * *Chats*
    * *Parse Mode*
    * *Updates*
    * *Bots*
    * *Buttons*
    * *Account*

2.7.1 TelegramClient

This is a summary of the methods and properties you will find at *TelegramClient*.

Auth

start	Starts the client (connects and logs in if necessary).
send_code_request	Sends the Telegram code needed to login to the given phone number.
sign_in	Logs in to Telegram to an existing user or bot account.
sign_up	Signs up to Telegram as a new user account.
log_out	Logs out Telegram and deletes the current `*.session` file.
edit_2fa	Changes the 2FA settings of the logged in user.

Base

connect	Connects to Telegram.
disconnect	Disconnects from Telegram.
is_connected	Returns `True` if the user has connected.
disconnected	Property with a `Future` that resolves upon disconnection.
loop	Property with the `asyncio` event loop used by this client.

Messages

send_message	Sends a message to the specified user, chat or channel.
edit_message	Edits the given message to change its text or media.
delete_messages	Deletes the given messages, optionally "for everyone".
forward_messages	Forwards the given messages to the specified entity.
iter_messages	Iterator over the messages for the given chat.
get_messages	Same as *iter_messages()*, but returns a *TotalList* instead.
pin_message	Pins or unpins a message in a chat.
send_read_acknowledge	Marks messages as read and optionally clears mentions.

Uploads

send_file	Sends message with the given file to the specified entity.
upload_file	Uploads a file to Telegram's servers, without sending it.

Downloads

download_media	Downloads the given media from a message object.

Table 5 – continued from previous page	
download_profile_photo	Downloads the profile photo from the given user, chat or channel.
download_file	Low-level method to download files from their input location.

Dialogs

iter_dialogs	Iterator over the dialogs (open conversations/subscribed channels).
get_dialogs	Same as *iter_dialogs()*, but returns a *TotalList* instead.
`archive`	
iter_drafts	Iterator over all open draft messages.
get_drafts	Same as *iter_drafts()*, but returns a list instead.
delete_dialog	Deletes a dialog (leaves a chat or channel).
conversation	Creates a *Conversation* with the given entity.

Users

get_me	Gets "me", the current User who is logged in.
is_bot	Return `True` if the signed-in user is a bot, `False` otherwise.
is_user_authorized	Returns `True` if the user is authorized (i.e.
get_entity	Turns the given entity into a valid Telegram User, Chat or Channel.
get_input_entity	Turns the given entity into its input entity version.
get_peer_id	Gets the ID for the given entity.

Chats

iter_participants	Iterator over the participants belonging to the specified chat.
get_participants	Same as *iter_participants()*, but returns a *TotalList* instead.
iter_admin_log	Iterator over the admin log for the specified channel.
get_admin_log	Same as *iter_admin_log()*, but returns a `list` instead.
iter_profile_photos	Iterator over a user's profile photos or a chat's photos.
get_profile_photos	Same as *iter_profile_photos()*, but returns a *TotalList* instead.
edit_admin	Edits admin permissions for someone in a chat.
edit_permissions	Edits user restrictions in a chat.
action	Returns a context-manager object to represent a "chat action".

Parse Mode

parse_mode	This property is the default parse mode used when sending messages.

Updates

on	Decorator used to *add_event_handler* more conveniently.
run_until_disconnected	Runs the event loop until the library is disconnected.
add_event_handler	Registers a new event handler callback.
remove_event_handler	Inverse operation of *add_event_handler()*.
list_event_handlers	Lists all registered event handlers.
catch_up	"Catches up" on the missed updates while the client was offline.

Bots

inline_query	Makes an inline query to the specified bot (e.g.

Buttons

build_reply_markup	Builds a ReplyInlineMarkup or ReplyKeyboardMarkup for the given buttons.

Account

takeout	Returns a *TelegramClient* which calls methods behind a takeout session.
end_takeout	Finishes the current takeout session.

2.8 Events Reference

Here you will find a quick summary of all the methods and properties that you can access when working with events.

You can access the client that creates this event by doing `event.client`, and you should view the description of the events to find out what arguments it allows on creation and its **attributes** (the properties will be shown here).

Important: Remember that **all events base** `ChatGetter`! Please see *FAQ* if you don't know what this means or the implications of it.

Contents

2.8.1 NewMessage

Occurs whenever a new text message or a message with media arrives.

Note: The new message event **should be treated as** a normal `Message`, with the following exceptions:

- `pattern_match` is the match object returned by `pattern=`.
- `message` is **not** the message string. It's the `Message` object.

Remember, this event is just a proxy over the message, so while you won't see its attributes and properties, you can still access them. Please see the full documentation for examples.

Full documentation for the `NewMessage`.

2.8.2 MessageEdited

Occurs whenever a message is edited. Just like `NewMessage`, you should treat this event as a `Message`.

Full documentation for the `MessageEdited`.

2.8.3 MessageDeleted

Occurs whenever a message is deleted. Note that this event isn't 100% reliable, since Telegram doesn't always notify the clients that a message was deleted.

It only has the `deleted_id` and `deleted_ids` attributes (in addition to the chat if the deletion happened in a channel).

Full documentation for the `MessageDeleted`.

2.8.4 MessageRead

Occurs whenever one or more messages are read in a chat.

Full documentation for the `MessageRead`.

inbox	`True` if you have read someone else's messages.

Continued on next page

Table 14 – continued from previous page

message_ids	The IDs of the messages **which contents'** were read.
get_messages	Returns the list of *Message* **which contents'** were read.
is_read	Returns True if the given message (or its ID) has been read.

2.8.5 ChatAction

Occurs whenever a user joins or leaves a chat, or a message is pinned.

Full documentation for the *ChatAction*.

added_by	The user who added users, if applicable (None otherwise).
kicked_by	The user who kicked users, if applicable (None otherwise).
user	The first user that takes part in this action (e.g.
input_user	Input version of the self.user property.
user_id	Returns the marked signed ID of the first user, if any.
users	A list of users that take part in this action (e.g.
input_users	Input version of the self.users property.
user_ids	Returns the marked signed ID of the users, if any.
respond	Responds to the chat action message (not as a reply).
reply	Replies to the chat action message (as a reply).
delete	Deletes the chat action message.
get_pinned_message	If new_pin is True, this returns the *Message* object that was pinned.
get_added_by	Returns *added_by* but will make an API call if necessary.
get_kicked_by	Returns *kicked_by* but will make an API call if necessary.
get_user	Returns *user* but will make an API call if necessary.
get_input_user	Returns *input_user* but will make an API call if necessary.
get_users	Returns *users* but will make an API call if necessary.
get_input_users	Returns *input_users* but will make an API call if necessary.

2.8.6 UserUpdate

Occurs whenever a user goes online, starts typing, etc.

Full documentation for the *UserUpdate*.

user	Alias for *sender*.
input_user	Alias for *input_sender*.
user_id	Alias for *sender_id*.
get_user	Alias for *get_sender*.
get_input_user	Alias for *get_input_sender*.

Continued on next page

Table 16 – continued from previous page

typing	`True` if the action is typing a message.
uploading	`True` if the action is uploading something.
recording	`True` if the action is recording something.
playing	`True` if the action is playing a game.
cancel	`True` if the action was cancelling other actions.
geo	`True` if what's being uploaded is a geo.
audio	`True` if what's being recorded/uploaded is an audio.
round	`True` if what's being recorded/uploaded is a round video.
video	`True` if what's being recorded/uploaded is an video.
contact	`True` if what's being uploaded (selected) is a contact.
document	`True` if what's being uploaded is document.
photo	`True` if what's being uploaded is a photo.
last_seen	Exact `datetime.datetime` when the user was last seen if known.
until	The `datetime.datetime` until when the user should appear online.
online	`True` if the user is currently online,
recently	`True` if the user was seen within a day.
within_weeks	`True` if the user was seen within 7 days.
within_months	`True` if the user was seen within 30 days.

2.8.7 CallbackQuery

Occurs whenever you sign in as a bot and a user clicks one of the inline buttons on your messages.

Full documentation for the *CallbackQuery*.

id	Returns the query ID.
message_id	Returns the message ID to which the clicked inline button belongs.
data	Returns the data payload from the original inline button.
chat_instance	Unique identifier for the chat where the callback occurred.
via_inline	Whether this callback was generated from an inline button sent via an inline query or not.
respond	Responds to the message (not as a reply).
reply	Replies to the message (as a reply).
edit	Edits the message.
delete	Deletes the message.
answer	Answers the callback query (and stops the loading circle).
get_message	Returns the message to which the clicked inline button belongs.

2.8.8 InlineQuery

Occurs whenever you sign in as a bot and a user sends an inline query such as `@bot query`.

Full documentation for the *InlineQuery*.

id	Returns the unique identifier for the query ID.
text	Returns the text the user used to make the inline query.
offset	The string the user's client used as an offset for the query.
geo	If the user location is requested when using inline mode and the user's device is able to send it, this will return the GeoPoint with the position of the user.
builder	Returns a new `InlineBuilder` instance.
answer	Answers the inline query with the given results.

2.8.9 Album

Occurs whenever you receive an entire album.

Full documentation for the `Album`.

grouped_id	The shared `grouped_id` between all the messages.
text	The message text of the first photo with a caption, formatted using the client's default parse mode.
raw_text	The raw message text of the first photo with a caption, ignoring any formatting.
is_reply	`True` if the album is a reply to some other message.
forward	The `Forward` information for the first message in the album if it was forwarded.
get_reply_message	The `Message` that this album is replying to, or `None`.
respond	Responds to the album (not as a reply).
reply	Replies to the first photo in the album (as a reply).
forward_to	Forwards the entire album.
edit	Edits the first caption or the message, or the first messages' caption if no caption is set, iff it's outgoing.
delete	Deletes the entire album.
mark_read	Marks the entire album as read.
pin	Pins the first photo in the album.

2.8.10 Raw

Raw events are not actual events. Instead, they are the raw Update object that Telegram sends. You normally shouldn't need these.

2.9 Objects Reference

This is the quick reference for those objects returned by client methods or other useful modules that the library has to offer. They are kept in a separate page to help finding and discovering them.

Remember that this page only shows properties and methods, **not attributes**. Make sure to open the full documentation to find out about the attributes.

Contents

2.9.1 ChatGetter

All events base `ChatGetter`, and some of the objects below do too, so it's important to know its methods.

`chat`	Returns the User, Chat or Channel where this object belongs to.
`input_chat`	This InputPeer is the input version of the chat where the message was sent.
`chat_id`	Returns the marked chat integer ID.
`is_private`	`True` if the message was sent as a private message.
`is_group`	True if the message was sent on a group or megagroup.
`is_channel`	`True` if the message was sent on a megagroup or channel.
`get_chat`	Returns `chat`, but will make an API call to find the chat unless it's already cached.
`get_input_chat`	Returns `input_chat`, but will make an API call to find the input chat unless it's already cached.

2.9.2 SenderGetter

Similar to `ChatGetter`, a `SenderGetter` is the same, but it works for senders instead.

`sender`	Returns the User or Channel that sent this object.
`input_sender`	This InputPeer is the input version of the user/channel who sent the message.
`sender_id`	Returns the marked sender integer ID, if present.

get_sender	Returns *sender*, but will make an API call to find the sender unless it's already cached.
get_input_sender	Returns *input_sender*, but will make an API call to find the input sender unless it's already cached.

2.9.3 Message

The *Message* type is very important, mostly because we are working with a library for a *messaging* platform, so messages are widely used: in events, when fetching history, replies, etc.

It bases *ChatGetter* and *SenderGetter*.

Properties

Note: We document *custom properties* here, not all the attributes of the *Message* (which is the information Telegram actually returns).

text	The message text, formatted using the client's default parse mode.
raw_text	The raw message text, ignoring any formatting.
is_reply	*True* if the message is a reply to some other message.
forward	The *Forward* information if this message is a forwarded message.
buttons	Returns a list of lists of *MessageButton*, if any.
button_count	Returns the total button count (sum of all *buttons* rows).
file	Returns a *File* wrapping the *photo* or *document* in this message.
photo	The Photo media in this message, if any.
document	The Document media in this message, if any.
web_preview	The WebPage media in this message, if any.
audio	The Document media in this message, if it's an audio file.
voice	The Document media in this message, if it's a voice note.
video	The Document media in this message, if it's a video.
video_note	The Document media in this message, if it's a video note.
gif	The Document media in this message, if it's a "gif".
sticker	The Document media in this message, if it's a sticker.
contact	The MessageMediaContact in this message, if it's a contact.
game	The Game media in this message, if it's a game.
geo	The GeoPoint media in this message, if it has a location.
invoice	The MessageMediaInvoice in this message, if it's an invoice.
poll	The MessageMediaPoll in this message, if it's a poll.

Table 22 – continued from previous page

venue	The MessageMediaVenue in this message, if it's a venue.
action_entities	Returns a list of entities that took part in this action.
via_bot	The bot User if the message was sent via said bot.
via_input_bot	Returns the input variant of *via_bot*.
client	Returns the *TelegramClient* that *patched* this message.

Methods

respond	Responds to the message (not as a reply).
reply	Replies to the message (as a reply).
forward_to	Forwards the message.
edit	Edits the message iff it's outgoing.
delete	Deletes the message.
get_reply_message	The `Message` that this message is replying to, or `None`.
click	Calls *button.click* on the specified button.
pin	Pins the message.
download_media	Downloads the media contained in the message, if any.
get_entities_text	Returns a list of `(markup entity, inner text)` (like bold or italics).
get_buttons	Returns *buttons* when that property fails (this is rarely needed).

2.9.4 File

The *File* type is a wrapper object returned by *Message.file*, and you can use it to easily access a document's attributes, such as its name, bot-API style file ID, etc.

id	The bot-API style `file_id` representing this file.
name	The file name of this document.
width	The width in pixels of this media if it's a photo or a video.
height	The height in pixels of this media if it's a photo or a video.
size	The size in bytes of this file.
duration	The duration in seconds of the audio or video.
title	The title of the song.
performer	The performer of the song.
emoji	A string with all emoji that represent the current sticker.
sticker_set	The InputStickerSet to which the sticker file belongs.

2.9.5 Conversation

The *Conversation* object is returned by the *client.conversation()* method to easily send and receive responses like a normal conversation.

It bases *ChatGetter*.

send_message	Sends a message in the context of this conversation.
send_file	Sends a file in the context of this conversation.
mark_read	Marks as read the latest received message if `message is None`.
get_response	Gets the next message that responds to a previous one.
get_reply	Gets the next message that explicitly replies to a previous one.
get_edit	Awaits for an edit after the last message to arrive.
wait_read	Awaits for the sent message to be marked as read.
wait_event	Waits for a custom event to occur.
cancel	Cancels the current conversation.

2.9.6 AdminLogEvent

The *AdminLogEvent* object is returned by the *client.iter_admin_log()* method to easily iterate over past "events" (deleted messages, edits, title changes, leaving members. . .)

These are all the properties you can find in it:

id	The ID of this event.
date	The date when this event occured.
user_id	The ID of the user that triggered this event.
action	The original ChannelAdminLogEventAction.
old	The old value from the event.
new	The new value present in the event.
changed_about	Whether the channel's about was changed or not.
changed_title	Whether the channel's title was changed or not.
changed_username	Whether the channel's username was changed or not.
changed_photo	Whether the channel's photo was changed or not.
changed_sticker_set	Whether the channel's sticker set was changed or not.
changed_message	Whether a message in this channel was edited or not.
deleted_message	Whether a message in this channel was deleted or not.
changed_admin	Whether the permissions for an admin in this channel changed or not.
changed_restrictions	Whether a message in this channel was edited or not.
changed_invites	Whether the invites in the channel were toggled or not.
joined	Whether `user` joined through the channel's public username or not.
joined_invite	Whether a new user joined through an invite link to the channel or not.
left	Whether `user` left the channel or not.
changed_hide_history	Whether hiding the previous message history for new members in the channel was toggled or not.
changed_signatures	Whether the message signatures in the channel were toggled or not.
changed_pin	Whether a new message in this channel was pinned or not.
changed_default_banned_rights	Whether the default banned rights were changed or not.
stopped_poll	Whether a poll was stopped or not.

2.9.7 Button

The *Button* class is used when you login as a bot account to send messages with reply markup, such as inline buttons or custom keyboards.

These are the static methods you can use to create instances of the markup:

inline	Creates a new inline button with some payload data in it.
switch_inline	Creates a new inline button to switch to inline query.
url	Creates a new inline button to open the desired URL on click.
text	Creates a new keyboard button with the given text.
request_location	Creates a new keyboard button to request the user's location on click.
request_phone	Creates a new keyboard button to request the user's phone on click.
clear	Clears all keyboard buttons after sending a message with this markup.
force_reply	Forces a reply to the message with this markup.

2.9.8 InlineResult

The *InlineResult* object is returned inside a list by the *client.inline_query()* method to make an inline query to a bot that supports being used in inline mode, such as @like.

Note that the list returned is in fact a *subclass* of a list called *InlineResults*, which, in addition of being a list (iterator, indexed access, etc.), has extra attributes and methods.

These are the constants for the types, properties and methods you can find the individual results:

ARTICLE	
PHOTO	
GIF	
VIDEO	
VIDEO_GIF	
AUDIO	
DOCUMENT	
LOCATION	
VENUE	
CONTACT	
GAME	
type	The always-present type of this result.
message	The always-present BotInlineMessage that will be sent if *click* is called on this result.
title	The title for this inline result.
description	The description for this inline result.
url	The URL present in this inline results.
photo	Returns either the WebDocument thumbnail for normal results or the Photo for media results.
document	Returns either the WebDocument content for normal results or the Document for media results.

Continued on next page

<div style="text-align:center">Table 28 – continued from previous page</div>

click	Clicks this result and sends the associated *message*.
download_media	Downloads the media in this result (if there is a document, the document will be downloaded; otherwise, the photo will if present).

2.9.9 Dialog

The *Dialog* object is returned when you call *client.iter_dialogs()*.

send_message	Sends a message to this dialog.
archive	Archives (or un-archives) this dialog.
delete	Deletes the dialog from your dialog list.

2.9.10 Draft

The *Draft* object is returned when you call *client.iter_drafts()*.

entity	The entity that belongs to this dialog (user, chat or channel).
input_entity	Input version of the entity.
get_entity	Returns *entity* but will make an API call if necessary.
get_input_entity	Returns *input_entity* but will make an API call if necessary.
text	The markdown text contained in the draft.
raw_text	The raw (text without formatting) contained in the draft.
is_empty	Convenience bool to determine if the draft is empty or not.
set_message	Changes the draft message on the Telegram servers.
send	Sends the contents of this draft to the dialog.
delete	Deletes this draft, and returns *True* on success.

2.9.11 Utils

The *telethon.utils* module has plenty of methods that make using the library a lot easier. Only the interesting ones will be listed here.

get_display_name	Gets the display name for the given User, Chat or Channel.
get_extension	Gets the corresponding extension for any Telegram media.
get_inner_text	Gets the inner text that's surrounded by the given entities.
get_peer_id	Convert the given peer into its marked ID by default.
resolve_id	Given a marked ID, returns the original ID and its Peer type.
pack_bot_file_id	Inverse operation for *resolve_bot_file_id*.

<div style="text-align:center">Continued on next page</div>

Table 31 – continued from previous page

resolve_bot_file_id	Given a Bot API-style *file_id*, returns the media it represents.
resolve_invite_link	Resolves the given invite link.

2.10 String-based Debugging

Debugging is *really* important. Telegram's API is really big and there is a lot of things that you should know. Such as, what attributes or fields does a result have? Well, the easiest thing to do is printing it:

```python
user = client.get_entity('Lonami')
print(user)
```

That will show a huge **string** similar to the following:

```
User(id=10885151, is_self=False, contact=False, mutual_contact=False, deleted=False,
→bot=False, bot_chat_history=False, bot_nochats=False, verified=False,
→restricted=False, min=False, bot_inline_geo=False, access_hash=123456789012345678,
→first_name='Lonami', last_name=None, username='Lonami', phone=None,
→photo=UserProfilePhoto(photo_id=123456789012345678, photo_small=FileLocation(dc_
→id=4, volume_id=1234567890, local_id=1234567890, secret=123456789012345678), photo_
→big=FileLocation(dc_id=4, volume_id=1234567890, local_id=1234567890,
→secret=123456789012345678)), status=UserStatusOffline(was_online=datetime.
→datetime(2018, 1, 2, 3, 4, 5, tzinfo=datetime.timezone.utc)), bot_info_version=None,
→ restriction_reason=None, bot_inline_placeholder=None, lang_code=None)
```

That's a lot of text. But as you can see, all the properties are there. So if you want the username you **don't use regex** or anything like splitting str(user) to get what you want. You just access the attribute you need:

```python
username = user.username
```

Can we get better than the shown string, though? Yes!

```python
print(user.stringify())
```

Will show a much better:

```
User(
    id=10885151,
    is_self=False,
    contact=False,
    mutual_contact=False,
    deleted=False,
    bot=False,
    bot_chat_history=False,
    bot_nochats=False,
    verified=False,
    restricted=False,
    min=False,
    bot_inline_geo=False,
    access_hash=123456789012345678,
    first_name='Lonami',
    last_name=None,
    username='Lonami',
    phone=None,
```

(continues on next page)

```
    photo=UserProfilePhoto(
        photo_id=123456789012345678,
        photo_small=FileLocation(
            dc_id=4,
            volume_id=123456789,
            local_id=123456789,
            secret=-123456789012345678
        ),
        photo_big=FileLocation(
            dc_id=4,
            volume_id=123456789,
            local_id=123456789,
            secret=123456789012345678
        )
    ),
    status=UserStatusOffline(
        was_online=datetime.datetime(2018, 1, 2, 3, 4, 5, tzinfo=datetime.timezone.
→utc)
    ),
    bot_info_version=None,
    restriction_reason=None,
    bot_inline_placeholder=None,
    lang_code=None
)
```

Now it's easy to see how we could get, for example, the `was_online` time. It's inside `status`:

```
online_at = user.status.was_online
```

You don't need to print everything to see what all the possible values can be. You can just search in http://tl.telethon.dev/.

Remember that you can use Python's isinstance to check the type of something. For example:

```
from telethon import types

if isinstance(user.status, types.UserStatusOffline):
    print(user.status.was_online)
```

2.11 Entities

The library widely uses the concept of "entities". An entity will refer to any User, Chat or Channel object that the API may return in response to certain methods, such as GetUsersRequest.

Note: When something "entity-like" is required, it means that you need to provide something that can be turned into an entity. These things include, but are not limited to, usernames, exact titles, IDs, Peer objects, or even entire User, Chat and Channel objects and even phone numbers **from people you have in your contact list**.

To "encounter" an ID, you would have to "find it" like you would in the normal app. If the peer is in your dialogs, you would need to *client.get_dialogs()*. If the peer is someone in a group, you would similarly *client.get_participants(group)*.

Once you have encountered an ID, the library will (by default) have saved their `access_hash` for you, which is needed to invoke most methods. This is why sometimes you might encounter this error when working with the library.

You should `except ValueError` and run code that you know should work to find the entity.

Contents

2.11.1 What is an Entity?

A lot of methods and requests require *entities* to work. For example, you send a message to an *entity*, get the username of an *entity*, and so on.

There are a lot of things that work as entities: usernames, phone numbers, chat links, invite links, IDs, and the types themselves. That is, you can use any of those when you see an "entity" is needed.

Note: Remember that the phone number must be in your contact list before you can use it.

You should use, **from better to worse**:

1. Input entities. For example, *event.input_chat*, *message.input_sender*, or caching an entity you will use a lot with `entity = await client.get_input_entity(...)`.

2. Entities. For example, if you had to get someone's username, you can just use `user` or `channel`. It will work. Only use this option if you already have the entity!

3. IDs. This will always look the entity up from the cache (the `*.session` file caches seen entities).

4. Usernames, phone numbers and links. The cache will be used too (unless you force a *client.get_entity()*), but may make a request if the username, phone or link has not been found yet.

In recent versions of the library, the following two are equivalent:

```python
async def handler(event):
    await client.send_message(event.sender_id, 'Hi')
    await client.send_message(event.input_sender, 'Hi')
```

If you need to be 99% sure that the code will work (sometimes it's simply impossible for the library to find the input entity), or if you will reuse the chat a lot, consider using the following instead:

```python
async def handler(event):
    # This method may make a network request to find the input sender.
    # Properties can't make network requests, so we need a method.
    sender = await event.get_input_sender()
    await client.send_message(sender, 'Hi')
    await client.send_message(sender, 'Hi')
```

2.11.2 Getting Entities

Through the use of the *Session Files*, the library will automatically remember the ID and hash pair, along with some extra information, so you're able to just do this:

```python
# Dialogs are the "conversations you have open".
# This method returns a list of Dialog, which
# has the .entity attribute and other information.
#
# This part is IMPORTANT, because it feels the entity cache.
dialogs = client.get_dialogs()

# All of these work and do the same.
lonami = client.get_entity('lonami')
lonami = client.get_entity('t.me/lonami')
lonami = client.get_entity('https://telegram.dog/lonami')

# Other kind of entities.
channel = client.get_entity('telegram.me/joinchat/AAAAAEkk2WdoDrB4-Q8-gg')
contact = client.get_entity('+34xxxxxxxxx')
friend  = client.get_entity(friend_id)

# Getting entities through their ID (User, Chat or Channel)
entity = client.get_entity(some_id)

# You can be more explicit about the type for said ID by wrapping
# it inside a Peer instance. This is recommended but not necessary.
from telethon.tl.types import PeerUser, PeerChat, PeerChannel

my_user    = client.get_entity(PeerUser(some_id))
my_chat    = client.get_entity(PeerChat(some_id))
my_channel = client.get_entity(PeerChannel(some_id))
```

Note: You **don't** need to get the entity before using it! Just let the library do its job. Use a phone from your contacts, username, ID or input entity (preferred but not necessary), whatever you already have.

All methods in the *TelegramClient* call `.get_input_entity()` prior to sending the request to save you from the hassle of doing so manually. That way, convenience calls such as `client.send_message('lonami', 'hi!')` become possible.

Every entity the library encounters (in any response to any call) will by default be cached in the `.session` file (an SQLite database), to avoid performing unnecessary API calls. If the entity cannot be found, additonal calls like ResolveUsernameRequest or GetContactsRequest may be made to obtain the required information.

2.11.3 Entities vs. Input Entities

Note: This section is informative, but worth reading. The library will transparently handle all of these details for you.

On top of the normal types, the API also make use of what they call their `Input*` versions of objects. The input version of an entity (e.g. InputPeerUser, InputChat, etc.) only contains the minimum information that's required from Telegram to be able to identify who you're referring to: a Peer's **ID** and **hash**. They are named like this because they are input parameters in the requests.

Entities' ID are the same for all user and bot accounts, however, the access hash is **different for each account**, so trying to reuse the access hash from one account in another will **not** work.

Sometimes, Telegram only needs to indicate the type of the entity along with their ID. For this purpose, Peer versions of the entities also exist, which just have the ID. You cannot get the hash out of them since you should not be needing it. The library probably has cached it before.

Peers are enough to identify an entity, but they are not enough to make a request with them use them. You need to know their hash before you can "use them", and to know the hash you need to "encounter" them, let it be in your dialogs, participants, message forwards, etc.

Note: You *can* use peers with the library. Behind the scenes, they are replaced with the input variant. Peers "aren't enough" on their own but the library will do some more work to use the right type.

As we just mentioned, API calls don't need to know the whole information about the entities, only their ID and hash. For this reason, another method, `client.get_input_entity()` is available. This will always use the cache while possible, making zero API calls most of the time. When a request is made, if you provided the full entity, e.g. an User, the library will convert it to the required InputPeer automatically for you.

You should always favour `client.get_input_entity()` **over** `client.get_entity()` for this reason! Calling the latter will always make an API call to get the most recent information about said entity, but invoking requests don't need this information, just the InputPeer. Only use `client.get_entity()` if you need to get actual information, like the username, name, title, etc. of the entity.

To further simplify the workflow, since the version `0.16.2` of the library, the raw requests you make to the API are also able to call `client.get_input_entity()` wherever needed, so you can even do things like:

```
client(SendMessageRequest('username', 'hello'))
```

The library will call the `.resolve()` method of the request, which will resolve `'username'` with the appropriated InputPeer. Don't worry if you don't get this yet, but remember some of the details here are important.

2.11.4 Full Entities

In addition to PeerUser, InputPeerUser, User (and its variants for chats and channels), there is also the concept of UserFull.

This full variant has additional information such as whether the user is blocked, its notification settings, the bio or about of the user, etc.

There is also messages.ChatFull which is the equivalent of full entities for chats and channels, with also the about section of the channel. Note that the `users` field only contains bots for the channel (so that clients can suggest commands to use).

You can get both of these by invoking GetFullUser, GetFullChat and GetFullChannel respectively.

2.11.5 Accessing Entities

Although it's explicitly noted in the documentation that messages *subclass* `ChatGetter` and `SenderGetter`, some people still don't get inheritance.

When the documentation says "Bases: `telethon.tl.custom.chatgetter.ChatGetter`" it means that the class you're looking at, *also* can act as the class it bases. In this case, `ChatGetter` knows how to get the *chat* where a thing belongs to.

So, a `Message` is a `ChatGetter`. That means you can do this:

```
message.is_private
message.chat_id
message.get_chat()
# ...etc
```

SenderGetter is similar:

```
message.user_id
message.get_input_user()
message.user
# ...etc
```

Quite a few things implement them, so it makes sense to reuse the code. For example, all events (except raw updates) implement *ChatGetter* since all events occur in some chat.

2.11.6 Summary

TL;DR; If you're here because of *"Could not find the input entity for"*, you must ask yourself "how did I find this entity through official applications"? Now do the same with the library. Use what applies:

```python
with client:
    # Does it have an username? Use it!
    entity = client.get_entity(username)

    # Do you have a conversation open with them? Get dialogs.
    client.get_dialogs()

    # Are they participant of some group? Get them.
    client.get_participants('TelethonChat')

    # Is the entity the original sender of a forwarded message? Get it.
    client.get_messages('TelethonChat', 100)

    # NOW you can use the ID, anywhere!
    entity = client.get_entity(123456)
    client.send_message(123456, 'Hi!')
```

Once the library has "seen" the entity, you can use their **integer** ID. You can't use entities from IDs the library hasn't seen. You must make the library see them *at least once* and disconnect properly. You know where the entities are and you must tell the library. It won't guess for you.

2.12 Updates in Depth

2.12.1 Properties vs. Methods

The event shown above acts just like a *custom.Message*, which means you can access all the properties it has, like .sender.

However events are different to other methods in the client, like *client.get_messages*. Events *may not* send information about the sender or chat, which means it can be None, but all the methods defined in the client always have this information so it doesn't need to be re-fetched. For this reason, you have get_ methods, which will make a network call if necessary.

In short, you should do this:

```
@client.on(events.NewMessage)
async def handler(event):
    # event.input_chat may be None, use event.get_input_chat()
    chat = await event.get_input_chat()
    sender = await event.get_sender()
    buttons = await event.get_buttons()

async def main():
    async for message in client.iter_messages('me', 10):
        # Methods from the client always have these properties ready
        chat = message.input_chat
        sender = message.sender
        buttons = message.buttons
```

Notice, properties (`message.sender`) don't need an `await`, but methods (`message.get_sender`) **do** need an `await`, and you should use methods in events for these properties that may need network.

2.12.2 Events Without the client

The code of your application starts getting big, so you decide to separate the handlers into different files. But how can you access the client from these files? You don't need to! Just `events.register` them:

```
# handlers/welcome.py
from telethon import events

@events.register(events.NewMessage('(?i)hello'))
async def handler(event):
    client = event.client
    await event.respond('Hey!')
    await client.send_message('me', 'I said hello to someone')
```

Registering events is a way of saying "this method is an event handler". You can use `telethon.events.is_handler` to check if any method is a handler. You can think of them as a different approach to Flask's blueprints.

It's important to note that this does **not** add the handler to any client! You never specified the client on which the handler should be used. You only declared that it is a handler, and its type.

To actually use the handler, you need to `client.add_event_handler` to the client (or clients) where they should be added to:

```
# main.py
from telethon import TelegramClient
import handlers.welcome

with TelegramClient(...) as client:
    client.add_event_handler(handlers.welcome.handler)
    client.run_until_disconnected()
```

This also means that you can register an event handler once and then add it to many clients without re-declaring the event.

2.12.3 Events Without Decorators

If for any reason you don't want to use `telethon.events.register`, you can explicitly pass the event handler to use to the mentioned `client.add_event_handler`:

```
from telethon import TelegramClient, events

async def handler(event):
    ...

with TelegramClient(...) as client:
    client.add_event_handler(handler, events.NewMessage)
    client.run_until_disconnected()
```

Similarly, you also have *client.remove_event_handler* and *client.list_event_handlers*.

The `event` argument is optional in all three methods and defaults to *events.Raw* for adding, and `None` when removing (so all callbacks would be removed).

Note: The `event` type is ignored in *client.add_event_handler* if you have used *telethon.events.register* on the `callback` before, since that's the point of using such method at all.

2.12.4 Stopping Propagation of Updates

There might be cases when an event handler is supposed to be used solitary and it makes no sense to process any other handlers in the chain. For this case, it is possible to raise a *telethon.events.StopPropagation* exception which will cause the propagation of the update through your handlers to stop:

```
from telethon.events import StopPropagation

@client.on(events.NewMessage)
async def _(event):
    # ... some conditions
    await event.delete()

    # Other handlers won't have an event to work with
    raise StopPropagation

@client.on(events.NewMessage)
async def _(event):
    # Will never be reached, because it is the second handler
    # in the chain.
    pass
```

Remember to check *Update Events* if you're looking for the methods reference.

2.12.5 Understanding asyncio

With `asyncio`, the library has several tasks running in the background. One task is used for sending requests, another task is used to receive them, and a third one is used to handle updates.

To handle updates, you must keep your script running. You can do this in several ways. For instance, if you are *not* running `asyncio`'s event loop, you should use *client.run_until_disconnected*:

```
import asyncio
from telethon import TelegramClient

client = TelegramClient(...)
```

```
...
client.run_until_disconnected()
```

Behind the scenes, this method is `await`'ing on the `client.disconnected` property, so the code above and the following are equivalent:

```python
import asyncio
from telethon import TelegramClient

client = TelegramClient(...)

async def main():
    await client.disconnected

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

You could also run `client.disconnected` until it completed.

But if you don't want to `await`, then you should know what you want to be doing instead! What matters is that you shouldn't let your script die. If you don't care about updates, you don't need any of this.

Notice that unlike `client.disconnected`, `client.run_until_disconnected` will handle `KeyboardInterrupt` with you. This method is special and can also be ran while the loop is running, so you can do this:

```python
async def main():
    await client.run_until_disconnected()

loop.run_until_complete(main())
```

2.12.6 Sequential Updates

If you need to process updates sequentially (i.e. not in parallel), you should set `sequential_updates=True` when creating the client:

```python
with TelegramClient(..., sequential_updates=True) as client:
    ...
```

2.13 Session Files

Contents

- *Session Files*
  - *What are Sessions?*
  - *Different Session Storage*
  - *Creating your Own Storage*
  - *String Sessions*

They are an important part for the library to be efficient, such as caching and handling your authorization key (or you would have to login every time!).

2.13.1 What are Sessions?

The first parameter you pass to the constructor of the *TelegramClient* is the `session`, and defaults to be the session name (or full path). That is, if you create a `TelegramClient('anon')` instance and connect, an `anon.session` file will be created in the working directory.

Note that if you pass a string it will be a file in the current working directory, although you can also pass absolute paths.

The session file contains enough information for you to login without re-sending the code, so if you have to enter the code more than once, maybe you're changing the working directory, renaming or removing the file, or using random names.

These database files using `sqlite3` contain the required information to talk to the Telegram servers, such as to which IP the client should connect, port, authorization key so that messages can be encrypted, and so on.

These files will by default also save all the input entities that you've seen, so that you can get information about a user or channel by just their ID. Telegram will **not** send their `access_hash` required to retrieve more information about them, if it thinks you have already seem them. For this reason, the library needs to store this information offline.

The library will by default too save all the entities (chats and channels with their name and username, and users with the phone too) in the session file, so that you can quickly access them by username or phone number.

If you're not going to work with updates, or don't need to cache the `access_hash` associated with the entities' ID, you can disable this by setting `client.session.save_entities = False`.

2.13.2 Different Session Storage

If you don't want to use the default SQLite session storage, you can also use one of the other implementations or implement your own storage.

While it's often not the case, it's possible that SQLite is slow enough to be noticeable, in which case you can also use a different storage. Note that this is rare and most people won't have this issue, but it's worth a mention.

To use a custom session storage, simply pass the custom session instance to *TelegramClient* instead of the session name.

Telethon contains three implementations of the abstract `Session` class:

- *MemorySession*: stores session data within memory.

- *SQLiteSession*: stores sessions within on-disk SQLite databases. Default.

- *StringSession*: stores session data within memory, but can be saved as a string.

You can import these `from telethon.sessions`. For example, using the *StringSession* is done as follows:

```python
from telethon.sync import TelegramClient
from telethon.sessions import StringSession

with TelegramClient(StringSession(string), api_id, api_hash) as client:
    ...  # use the client

    # Save the string session as a string; you should decide how
    # you want to save this information (over a socket, remote
    # database, print it and then paste the string in the code,
```

(continues on next page)

```
    # etc.); the advantage is that you don't need to save it
    # on the current disk as a separate file, and can be reused
    # anywhere else once you log in.
    string = client.session.save()

# Note that it's also possible to save any other session type
# as a string by using ``StringSession.save(session_instance)``:
client = TelegramClient('sqlite-session', api_id, api_hash)
string = StringSession.save(client.session)
```

There are other community-maintained implementations available:

- SQLAlchemy: stores all sessions in a single database via SQLAlchemy.

- Redis: stores all sessions in a single Redis data store.

2.13.3 Creating your Own Storage

The easiest way to create your own storage implementation is to use *MemorySession* as the base and check out how *SQLiteSession* or one of the community-maintained implementations work. You can find the relevant Python files under the sessions/ directory in the Telethon's repository.

After you have made your own implementation, you can add it to the community-maintained session implementation list above with a pull request.

2.13.4 String Sessions

StringSession are a convenient way to embed your login credentials directly into your code for extremely easy portability, since all they take is a string to be able to login without asking for your phone and code (or faster start if you're using a bot token).

The easiest way to generate a string session is as follows:

```
from telethon.sync import TelegramClient
from telethon.sessions import StringSession

with TelegramClient(StringSession(), api_id, api_hash) as client:
    print(client.session.save())
```

Think of this as a way to export your authorization key (what's needed to login into your account). This will print a string in the standard output (likely your terminal).

> **Warning: Keep this string safe!** Anyone with this string can use it to login into your account and do anything they want to to do.
>
> This is similar to leaking your *.session files online, but it is easier to leak a string than it is to leak a file.

Once you have the string (which is a bit long), load it into your script somehow. You can use a normal text file and open(...).read() it or you can save it in a variable directly:

```
string = '1aaNk8EX-YRfwoRsebUkugFvht6DUPi_Q25UOCzOAqzc...'
with TelegramClient(StringSession(string), api_id, api_hash) as client:
    client.send_message('me', 'Hi')
```

These strings are really convenient for using in places like Heroku since their ephemeral filesystem will delete external files once your application is over.

2.14 The Full API

Important: While you have access to this, you should always use the friendly methods listed on *Client Reference* unless you have a better reason not to, like a method not existing or you wanting more control.

The *TelegramClient* doesn't offer a method for every single request the Telegram API supports. However, it's very simple to *call* or *invoke* any request. Whenever you need something, don't forget to check the documentation and look for the method you need. There you can go through a sorted list of everything you can do.

Note: The reason to keep both https://tl.telethon.dev and this documentation alive is that the former allows instant search results as you type, and a "Copy import" button. If you like namespaces, you can also do `from telethon.tl import types, functions`. Both work.

Important: All the examples in this documentation assume that you have `from telethon import sync` or `import telethon.sync` for the sake of simplicity and that you understand what it does (see *Compatibility and Convenience* for more). Simply add either line at the beginning of your project and it will work.

You should also refer to the documentation to see what the objects (constructors) Telegram returns look like. Every constructor inherits from a common type, and that's the reason for this distinction.

Say `client.send_message()` didn't exist, we could use the search to look for "message". There we would find SendMessageRequest, which we can work with.

Every request is a Python class, and has the parameters needed for you to invoke it. You can also call `help(request)` for information on what input parameters it takes. Remember to "Copy import to the clipboard", or your script won't be aware of this class! Now we have:

```python
from telethon.tl.functions.messages import SendMessageRequest
```

If you're going to use a lot of these, you may do:

```python
from telethon.tl import types, functions
# We now have access to 'functions.messages.SendMessageRequest'
```

We see that this request must take at least two parameters, a `peer` of type InputPeer, and a `message` which is just a Python `str`ing.

How can we retrieve this InputPeer? We have two options. We manually construct one, for instance:

```python
from telethon.tl.types import InputPeerUser

peer = InputPeerUser(user_id, user_hash)
```

Or we call `client.get_input_entity()`:

```python
import telethon.sync
peer = client.get_input_entity('someone')
```

When you're going to invoke an API method, most require you to pass an InputUser, InputChat, or so on, this is why using `client.get_input_entity()` is more straightforward (and often immediate, if you've seen the user before, know their ID, etc.). If you also **need** to have information about the whole user, use `client.get_entity()` instead:

```
entity = client.get_entity('someone')
```

In the later case, when you use the entity, the library will cast it to its "input" version for you. If you already have the complete user and want to cache its input version so the library doesn't have to do this every time its used, simply call `telethon.utils.get_input_peer`:

```
from telethon import utils
peer = utils.get_input_peer(entity)
```

Note: Since `v0.16.2` this is further simplified. The `Request` itself will call `client.get_input_entity` for you when required, but it's good to remember what's happening.

After this small parenthesis about `client.get_entity` versus `client.get_input_entity()`, we have everything we need. To invoke our request we do:

```
result = client(SendMessageRequest(peer, 'Hello there!'))
```

Message sent! Of course, this is only an example. There are over 250 methods available as of layer 80, and you can use every single of them as you wish. Remember to use the right types! To sum up:

```
result = client(SendMessageRequest(
    client.get_input_entity('username'), 'Hello there!'
))
```

This can further be simplified to:

```
result = client(SendMessageRequest('username', 'Hello there!'))
# Or even
result = client(SendMessageRequest(PeerChannel(id), 'Hello there!'))
```

Note: Note that some requests have a "hash" parameter. This is **not** your `api_hash`! It likely isn't your self-user `.access_hash` either.

It's a special hash used by Telegram to only send a difference of new data that you don't already have with that request, so you can leave it to 0, and it should work (which means no hash is known yet).

For those requests having a "limit" parameter, you can often set it to zero to signify "return default amount". This won't work for all of them though, for instance, in "messages.search" it will actually return 0 items.

2.14.1 Requests in Parallel

The library will automatically merge outgoing requests into a single *container*. Telegram's API supports sending multiple requests in a single container, which is faster because it has less overhead and the server can run them without waiting for others. You can also force using a container manually:

```
async def main():
```

```
    # Letting the library do it behind the scenes
    await asyncio.wait([
        client.send_message('me', 'Hello'),
        client.send_message('me', ','),
        client.send_message('me', 'World'),
        client.send_message('me', '.')
    ])

    # Manually invoking many requests at once
    await client([
        SendMessageRequest('me', 'Hello'),
        SendMessageRequest('me', ', '),
        SendMessageRequest('me', 'World'),
        SendMessageRequest('me', '.')
    ])
```

Note that you cannot guarantee the order in which they are run. Try running the above code more than one time. You will see the order in which the messages arrive is different.

If you use the raw API (the first option), you can use `ordered` to tell the server that it should run the requests sequentially. This will still be faster than going one by one, since the server knows all requests directly:

```
client([
    SendMessageRequest('me', 'Hello'),
    SendMessageRequest('me', ', '),
    SendMessageRequest('me', 'World'),
    SendMessageRequest('me', '.')
], ordered=True)
```

If any of the requests fails with a Telegram error (not connection errors or any other unexpected events), the library will raise *telethon.errors.common.MultiError*. You can `except` this and still access the successful results:

```
from telethon.errors import MultiError

try:
    client([
        SendMessageRequest('me', 'Hello'),
        SendMessageRequest('me', ''),
        SendMessageRequest('me', 'World')
    ], ordered=True)
except MultiError as e:
    # The first and third requests worked.
    first = e.results[0]
    third = e.results[2]
    # The second request failed.
    second = e.exceptions[1]
```

2.15 RPC Errors

RPC stands for Remote Procedure Call, and when the library raises a `RPCError`, it's because you have invoked some of the API methods incorrectly (wrong parameters, wrong permissions, or even something went wrong on Telegram's server). All the errors are available in *API Errors*, but some examples are:

- `FloodWaitError` (420), the same request was repeated many times. Must wait `.seconds` (you can access this attribute). For example:

```
...
from telethon import errors

try:
    print(client.get_messages(chat)[0].text)
except errors.FloodWaitError as e:
    print('Have to sleep', e.seconds, 'seconds')
    time.sleep(e.seconds)
```

- `SessionPasswordNeededError`, if you have setup two-steps verification on Telegram.

- `CdnFileTamperedError`, if the media you were trying to download from a CDN has been altered.

- `ChatAdminRequiredError`, you don't have permissions to perform said operation on a chat or channel. Try avoiding filters, i.e. when searching messages.

The generic classes for different error codes are:

- `InvalidDCError` (303), the request must be repeated on another DC.

- `BadRequestError` (400), the request contained errors.

- `UnauthorizedError` (401), the user is not authorized yet.

- `ForbiddenError` (403), privacy violation error.

- `NotFoundError` (404), make sure you're invoking `Request`'s!

If the error is not recognised, it will only be an `RPCError`.

You can refer to all errors from Python through the `telethon.errors` module. If you don't know what attributes they have, try printing their dir (like `print(dir(e))`).

2.15.1 Avoiding Limits

Don't spam. You won't get `FloodWaitError` or your account banned or deleted if you use the library *for legit use cases*. Make cool tools. Don't spam! Nobody knows the exact limits for all requests since they depend on a lot of factors, so don't bother asking.

Still, if you do have a legit use case and still get those errors, the library will automatically sleep when they are smaller than 60 seconds by default. You can set different "auto-sleep" thresholds:

```
client.flood_sleep_threshold = 0  # Don't auto-sleep
client.flood_sleep_threshold = 24 * 60 * 60  # Sleep always
```

You can also except it and act as you prefer:

```
from telethon.errors import FloodWaitError
try:
    ...
except FloodWaitError as e:
    print('Flood waited for', e.seconds)
    quit(1)
```

VoIP numbers are very limited, and some countries are more limited too.

2.16 HTTP Bot API vs MTProto

Telethon is more than just another viable alternative when developing 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.

Contents

- *HTTP Bot API vs MTProto*
    - *What is Bot API?*
    - *What is MTProto?*
    - *Advantages of MTProto over Bot API*
    - *Migrating from Bot API to Telethon*
        * *Migrating from python-telegram-bot*
        * *Migrating from pyTelegramBotAPI*
        * *Migrating from aiogram*
        * *Migrating from dumbot*

2.16.1 What is Bot API?

The Telegram Bot API, also known as HTTP Bot API and 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 which translates your requests to it into MTProto calls through tdlib, their bot backend.

2.16.2 What is MTProto?

MTProto is Telegram's own protocol to communicate with their API when you connect to their servers.

Telethon is an alternative MTProto-based backend written entirely in Python and much easier to setup and use.

Both official applications and third-party clients (like your own applications) logged in as either user or bots **can use MTProto** to communicate directly with Telegram's API (which is not the HTTP bot API).

When we talk about MTProto, we often mean "MTProto-based clients".

2.16.3 Advantages of MTProto over Bot API

MTProto clients (like Telethon) connect directly to Telegram's servers, which means there is no HTTP connection, no "polling" or "web hooks". This means **less overhead**, since the protocol used between you and the server is much more compact than HTTP requests with responses in wasteful JSON.

Since there is a direct connection to Telegram's servers, even if their Bot API endpoint is down, you can still have connection to Telegram directly.

Using a MTProto client, you are also not limited to the public API that they expose, and instead, **you have full control** of what your bot can do. Telethon offers you all the power with often **much easier usage** than any of the available Python Bot API wrappers.

If your application ever needs user features because bots cannot do certain things, you will be able to easily login as a user and even keep your bot without having to learn a new library.

If less overhead and full control didn't convince you to use Telethon yet, check out the repository HTTP Bot API vs MTProto comparison with a more exhaustive and up-to-date list of differences.

2.16.4 Migrating from Bot API to Telethon

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. It's never too late to migrate to Telethon!

If you were using an asynchronous library like aiohttp or a wrapper like aiogram or dumbot, it will be even easier, because Telethon is also an asynchronous library.

Next, we will see some examples from the most popular libraries.

Migrating from python-telegram-bot

Let's take their echobot2.py example and shorten it a bit:

```python
from telegram.ext import Updater, CommandHandler, MessageHandler, Filters

def start(update, context):
    """Send a message when the command /start is issued."""
    update.message.reply_text('Hi!')

def echo(update, context):
    """Echo the user message."""
    update.message.reply_text(update.message.text)

def main():
    """Start the bot."""
    updater = Updater("TOKEN")
    dp = updater.dispatcher
    dp.add_handler(CommandHandler("start", start))
    dp.add_handler(MessageHandler(Filters.text, echo))

    updater.start_polling()

    updater.idle()

if __name__ == '__main__':
    main()
```

After using Telethon:

```python
from telethon import TelegramClient, events

bot = TelegramClient('bot', 11111, 'a1b2c3d4').start(bot_token='TOKEN')

@bot.on(events.NewMessage(pattern='/start'))
async def start(event):
```

(continues on next page)

```python
    """Send a message when the command /start is issued."""
    await event.respond('Hi!')
    raise events.StopPropagation


@bot.on(events.NewMessage)
async def echo(event):
    """Echo the user message."""
    await event.respond(event.text)


def main():
    """Start the bot."""
    bot.run_until_disconnected()


if __name__ == '__main__':
    main()
```

Key differences:

- The recommended way to do it imports less things.

- All handlers trigger by default, so we need `events.StopPropagation`.

- Adding handlers, responding and running is a lot less verbose.

- Telethon needs `async def` and `await`.

- The `bot` isn't hidden away by `Updater` or `Dispatcher`.

Migrating from pyTelegramBotAPI

Let's show another echobot from their README:

```python
import telebot

bot = telebot.TeleBot("TOKEN")


@bot.message_handler(commands=['start'])
def send_welcome(message):
    bot.reply_to(message, "Howdy, how are you doing?")


@bot.message_handler(func=lambda m: True)
def echo_all(message):
    bot.reply_to(message, message.text)


bot.polling()
```

Now we rewrite it to use Telethon:

```python
from telethon import TelegramClient, events

bot = TelegramClient('bot', 11111, 'a1b2c3d4').start(bot_token='TOKEN')


@bot.on(events.NewMessage(pattern='/start'))
async def send_welcome(event):
    await event.reply('Howdy, how are you doing?')


@bot.on(events.NewMessage)
```

```python
async def echo_all(event):
    await event.reply(event.text)

bot.run_until_disconnected()
```

Key differences:

- Instead of doing `bot.reply_to(message)`, we can do `event.reply`. Note that the `event` behaves just like their `message`.

- Telethon also supports `func=lambda m:   True`, but it's not necessary.

Migrating from aiogram

From their GitHub:

```python
from aiogram import Bot, Dispatcher, executor, types

API_TOKEN = 'BOT TOKEN HERE'

# Initialize bot and dispatcher
bot = Bot(token=API_TOKEN)
dp = Dispatcher(bot)

@dp.message_handler(commands=['start'])
async def send_welcome(message: types.Message):
    """
    This handler will be called when client send `/start` command.
    """
    await message.reply("Hi!\nI'm EchoBot!\nPowered by aiogram.")

@dp.message_handler(regexp='(^cat[s]?$|puss)')
async def cats(message: types.Message):
    with open('data/cats.jpg', 'rb') as photo:
        await bot.send_photo(message.chat.id, photo, caption='Cats is here ',
                             reply_to_message_id=message.message_id)

@dp.message_handler()
async def echo(message: types.Message):
    await bot.send_message(message.chat.id, message.text)

if __name__ == '__main__':
    executor.start_polling(dp, skip_updates=True)
```

After rewrite:

```python
from telethon import TelegramClient, events

# Initialize bot and... just the bot!
bot = TelegramClient('bot', 11111, 'a1b2c3d4').start(bot_token='TOKEN')

@bot.on(events.NewMessage(pattern='/start'))
async def send_welcome(event):
    await event.reply('Howdy, how are you doing?')

@bot.on(events.NewMessage(pattern='(^cat[s]?$|puss)'))
```

```python
async def cats(event):
    await event.reply('Cats is here ', file='data/cats.jpg')

@bot.on(events.NewMessage)
async def echo_all(event):
    await event.reply(event.text)


if __name__ == '__main__':
    bot.run_until_disconnected()
```

Key differences:

- Telethon offers convenience methods to avoid retyping `bot.send_photo(message.chat.id, ...)` all the time, and instead let you type `event.reply`.

- Sending files is **a lot** easier. The methods for sending photos, documents, audios, etc. are all the same!

Migrating from dumbot

Showcasing their subclassing example:

```python
from dumbot import Bot

class Subbot(Bot):
    async def init(self):
        self.me = await self.getMe()

    async def on_update(self, update):
        await self.sendMessage(
            chat_id=update.message.chat.id,
            text='i am {}'.format(self.me.username)
        )

Subbot(token).run()
```

After rewriting:

```python
from telethon import TelegramClient, events

class Subbot(TelegramClient):
    def __init__(self, *a, **kw):
        await super().__init__(*a, **kw)
        self.add_event_handler(self.on_update, events.NewMessage)

    async def connect():
        await super().connect()
        self.me = await self.get_me()

    async def on_update(event):
        await event.reply('i am {}'.format(self.me.username))

bot = Subbot('bot', 11111, 'a1b2c3d4').start(bot_token='TOKEN')
bot.run_until_disconnected()
```

Key differences:

- Telethon method names are `snake_case`.

- dumbot does not offer friendly methods like `update.reply`.

- Telethon does not have an implicit `on_update` handler, so we need to manually register one.

2.17 Mastering asyncio

Contents

- *Mastering asyncio*
    - *What's asyncio?*
    - *Why asyncio?*
    - *What are asyncio basics?*
    - *What does telethon.sync do?*
    - *What are async, await and coroutines?*
    - *Can I use threads?*
    - *client.run_until_disconnected() blocks!*
    - *What else can asyncio do?*
    - *Why does client.start() work outside async?*
    - *Where can I read more?*

2.17.1 What's asyncio?

`asyncio` is a Python 3's built-in library. This means it's already installed if you have Python 3. Since Python 3.5, it is convenient to work with asynchronous code. Before (Python 3.4) we didn't have `async` or `await`, but now we do.

`asyncio` stands for *Asynchronous Input Output*. This is a very powerful concept to use whenever you work IO. Interacting with the web or external APIs such as Telegram's makes a lot of sense this way.

2.17.2 Why asyncio?

Asynchronous IO makes a lot of sense in a library like Telethon. You send a request to the server (such as "get some message"), and thanks to `asyncio`, your code won't block while a response arrives.

The alternative would be to spawn a thread for each update so that other code can run while the response arrives. That is *a lot* more expensive.

The code will also run faster, because instead of switching back and forth between the OS and your script, your script can handle it all. Avoiding switching saves quite a bit of time, in Python or any other language that supports asynchronous IO. It will also be cheaper, because tasks are smaller than threads, which are smaller than processes.

2.17.3 What are asyncio basics?

```python
# First we need the asyncio library
import asyncio

# Then we need a loop to work with
loop = asyncio.get_event_loop()

# We also need something to run
async def main():
    for char in 'Hello, world!\n':
        print(char, end='', flush=True)
        await asyncio.sleep(0.2)

# Then, we need to run the loop with a task
loop.run_until_complete(main())
```

2.17.4 What does telethon.sync do?

The moment you import any of these:

```python
from telethon import sync, ...
# or
from telethon.sync import ...
# or
import telethon.sync
```

The `sync` module rewrites most `async def` methods in Telethon to something similar to this:

```python
def new_method():
    result = original_method()
    if loop.is_running():
        # the loop is already running, return the await-able to the user
        return result
    else:
        # the loop is not running yet, so we can run it for the user
        return loop.run_until_complete(result)
```

That means you can do this:

```python
print(client.get_me().username)
```

Instead of this:

```python
import asyncio
loop = asyncio.get_event_loop()
me = loop.run_until_complete(client.get_me())
print(me.username)
```

As you can see, it's a lot of boilerplate and noise having to type `run_until_complete` all the time, so you can let the magic module to rewrite it for you. But notice the comment above: it won't run the loop if it's already running, because it can't. That means this:

```python
async def main():
    # 3. the loop is running here
    print(
        client.get_me()  # 4. this will return a coroutine!
```

```
        .username  # 5. this fails, coroutines don't have usernames
    )

loop.run_until_complete(  # 2. run the loop and the ``main()`` coroutine
    main()  # 1. calling ``async def`` "returns" a coroutine
)
```

Will fail. So if you're inside an `async def`, then the loop is running, and if the loop is running, you must `await` things yourself:

```
async def main():
    print((await client.get_me()).username)

loop.run_until_complete(main())
```

2.17.5 What are async, await and coroutines?

The `async` keyword lets you define asynchronous functions, also known as coroutines, and also iterate over asynchronous loops or use `async with`:

```
import asyncio

async def main():
    # ^ this declares the main() coroutine function

    async with client:
        # ^ this is an asynchronous with block

        async for message in client.iter_messages(chat):
            # ^ this is a for loop over an asynchronous generator

            print(message.sender.username)

loop = asyncio.get_event_loop()
# ^ this assigns the default event loop from the main thread to a variable

loop.run_until_complete(main())
# ^ this runs the *entire* loop until the main() function finishes.
#   While the main() function does not finish, the loop will be running.
#   While the loop is running, you can't run it again.
```

The `await` keyword blocks the *current* task, and the loop can run other tasks. Tasks can be thought of as "threads", since many can run concurrently:

```
import asyncio

async def hello(delay):
    await asyncio.sleep(delay)  # await tells the loop this task is "busy"
    print('hello')  # eventually the loop resumes the code here

async def world(delay):
    # the loop decides this method should run first
    await asyncio.sleep(delay)  # await tells the loop this task is "busy"
    print('world')  # eventually the loop finishes all tasks
```

```python
loop = asyncio.get_event_loop()  # get the default loop for the main thread
loop.create_task(world(2))  # create the world task, passing 2 as delay
loop.create_task(hello(delay=1))  # another task, but with delay 1
try:
    # run the event loop forever; ctrl+c to stop it
    # we could also run the loop for three seconds:
    #     loop.run_until_complete(asyncio.sleep(3))
    loop.run_forever()
except KeyboardInterrupt:
    pass
```

The same example, but without the comment noise:

```python
import asyncio

async def hello(delay):
    await asyncio.sleep(delay)
    print('hello')

async def world(delay):
    await asyncio.sleep(delay)
    print('world')

loop = asyncio.get_event_loop()
loop.create_task(world(2))
loop.create_task(hello(1))
loop.run_until_complete(asyncio.sleep(3))
```

2.17.6 Can I use threads?

Yes, you can, but you must understand that the loops themselves are not thread safe. and you must be sure to know what is happening. You may want to create a loop in a new thread and make sure to pass it to the client:

```python
import asyncio
import threading

def go():
    loop = asyncio.new_event_loop()
    client = TelegramClient(..., loop=loop)
    ...

threading.Thread(target=go).start()
```

Generally, **you don't need threads** unless you know what you're doing. Just create another task, as shown above. If you're using the Telethon with a library that uses threads, you must be careful to use `threading.Lock` whenever you use the client, or enable the compatible mode. For that, see *Compatibility and Convenience*.

You may have seen this error:

```
RuntimeError: There is no current event loop in thread 'Thread-1'.
```

It just means you didn't create a loop for that thread, and if you don't pass a loop when creating the client, it uses `asyncio.get_event_loop()`, which only works in the main thread.

2.17.7 client.run_until_disconnected() blocks!

All of what *client.run_until_disconnected()* does is run the `asyncio`'s event loop until the client is disconnected. That means *the loop is running*. And if the loop is running, it will run all the tasks in it. So if you want to run *other* code, create tasks for it:

```python
from datetime import datetime

async def clock():
    while True:
        print('The time:', datetime.now())
        await asyncio.sleep(1)

loop.create_task(clock())
...
client.run_until_disconnected()
```

This creates a task for a clock that prints the time every second. You don't need to use *client.run_until_disconnected()* either! You just need to make the loop is running, somehow. `loop.run_forever()` and `loop.run_until_complete()` can also be used to run the loop, and Telethon will be happy with any approach.

Of course, there are better tools to run code hourly or daily, see below.

2.17.8 What else can asyncio do?

Asynchronous IO is a really powerful tool, as we've seen. There are plenty of other useful libraries that also use `asyncio` and that you can integrate with Telethon.

- aiohttp is like the infamous requests but asynchronous.

- quart is an asynchronous alternative to Flask.

- aiocron lets you schedule things to run things at a desired time, or run some tasks hourly, daily, etc.

And of course, asyncio itself! It has a lot of methods that let you do nice things. For example, you can run requests in parallel:

```python
async def main():
    last, sent, download_path = await asyncio.gather(
        client.get_messages('TelethonChat', 10),
        client.send_message('TelethonOfftopic', 'Hey guys!'),
        client.download_profile_photo('TelethonChat')
    )

loop.run_until_complete(main())
```

This code will get the 10 last messages from @TelethonChat, send one to @TelethonOfftopic, and also download the profile photo of the main group. `asyncio` will run all these three tasks at the same time. You can run all the tasks you want this way.

A different way would be:

```python
loop.create_task(client.get_messages('TelethonChat', 10))
loop.create_task(client.send_message('TelethonOfftopic', 'Hey guys!'))
loop.create_task(client.download_profile_photo('TelethonChat'))
```

They will run in the background as long as the loop is running too.

You can also start an asyncio server in the main script, and from another script, connect to it to achieve Inter-Process Communication. You can get as creative as you want. You can program anything you want. When you use a library, you're not limited to use only its methods. You can combine all the libraries you want. People seem to forget this simple fact!

2.17.9 Why does client.start() work outside async?

Because it's so common that it's really convenient to offer said functionality by default. This means you can set up all your event handlers and start the client without worrying about loops at all.

Using the client in a `with` block, `start`, `run_until_disconnected`, and `disconnect` all support this.

2.17.10 Where can I read more?

Check out my blog post about `asyncio`, which has some more examples and pictures to help you understand what happens when the loop runs.

2.18 A Word of Warning

Full API is **not** how you are intended to use the library. You **should** always prefer the *Client Reference*. However, not everything is implemented as a friendly method, so full API is your last resort.

If you select a method in *Client Reference*, you will most likely find an example for that method. This is how you are intended to use the library.

Full API **will** break between different minor versions of the library, since Telegram changes very often. The friendly methods will be kept compatible between major versions.

If you need to see real-world examples, please refer to *Projects using Telethon*.

2.19 Working with Chats and Channels

Note: These examples assume you have read *The Full API*.

Contents

> – *Kicking a member*
>
> – *Increasing View Count in a Channel*

2.19.1 Joining a chat or channel

Note that Chat are normal groups, and Channel are a special form of Chat, which can also be super-groups if their `megagroup` member is `True`.

2.19.2 Joining a public channel

Once you have the *entity* of the channel you want to join to, you can make use of the JoinChannelRequest to join such channel:

```python
from telethon.tl.functions.channels import JoinChannelRequest
client(JoinChannelRequest(channel))

# In the same way, you can also leave such channel
from telethon.tl.functions.channels import LeaveChannelRequest
client(LeaveChannelRequest(input_channel))
```

For more on channels, check the channels namespace.

2.19.3 Joining a private chat or channel

If all you have is a link like this one: `https://t.me/joinchat/AAAAAFFszQPyPEZ7wgxLtd`, you already have enough information to join! The part after the `https://t.me/joinchat/`, this is, `AAAAAFFszQPyPEZ7wgxLtd` on this example, is the `hash` of the chat or channel. Now you can use ImportChat-InviteRequest as follows:

```python
from telethon.tl.functions.messages import ImportChatInviteRequest
updates = client(ImportChatInviteRequest('AAAAAEHbEkejzxUjAUCfYg'))
```

2.19.4 Adding someone else to such chat or channel

If you don't want to add yourself, maybe because you're already in, you can always add someone else with the AddChatUserRequest, which use is very straightforward, or InviteToChannelRequest for channels:

```python
# For normal chats
from telethon.tl.functions.messages import AddChatUserRequest

# Note that ``user_to_add`` is NOT the name of the parameter.
# It's the user you want to add (``user_id=user_to_add``).
client(AddChatUserRequest(
    chat_id,
    user_to_add,
    fwd_limit=10   # Allow the user to see the 10 last messages
))

# For channels (which includes megagroups)
from telethon.tl.functions.channels import InviteToChannelRequest
```

<div align="right">(continues on next page)</div>

```
client(InviteToChannelRequest(
    channel,
    [users_to_add]
))
```

2.19.5 Checking a link without joining

If you don't need to join but rather check whether it's a group or a channel, you can use the CheckChatInviteRequest, which takes in the hash of said channel or group.

2.19.6 Admin Permissions

Giving or revoking admin permissions can be done with the EditAdminRequest:

```python
from telethon.tl.functions.channels import EditAdminRequest
from telethon.tl.types import ChatAdminRights

# You need both the channel and who to grant permissions
# They can either be channel/user or input channel/input user.
#
# ChatAdminRights is a list of granted permissions.
# Set to True those you want to give.
rights = ChatAdminRights(
    post_messages=None,
    add_admins=None,
    invite_users=None,
    change_info=True,
    ban_users=None,
    delete_messages=True,
    pin_messages=True,
    invite_link=None,
    edit_messages=None
)
# Equivalent to:
#     rights = ChatAdminRights(
#         change_info=True,
#         delete_messages=True,
#         pin_messages=True
#     )

# Once you have a ChatAdminRights, invoke it
client(EditAdminRequest(channel, user, rights))

# User will now be able to change group info, delete other people's
# messages and pin messages.
#
# In a normal chat, you should do this instead:
from telethon.tl.functions.messages import EditChatAdminRequest

client(EditChatAdminRequest(chat_id, user, is_admin=True))
```

Note: Thanks to @Kyle2142 for pointing out that you **cannot** set all parameters to True to give a user full permis-

sions, as not all permissions are related to both broadcast channels/megagroups.

E.g. trying to set `post_messages=True` in a megagroup will raise an error. It is recommended to always use keyword arguments, and to set only the permissions the user needs. If you don't need to change a permission, it can be omitted (full list here).

2.19.7 Restricting Users

Similar to how you give or revoke admin permissions, you can edit the banned rights of a user through Edit-BannedRequest and its parameter ChatBannedRights:

```python
from telethon.tl.functions.channels import EditBannedRequest
from telethon.tl.types import ChatBannedRights

from datetime import datetime, timedelta

# Restricting a user for 7 days, only allowing view/send messages.
#
# Note that it's "reversed". You must set to `True` the permissions
# you want to REMOVE, and leave as `None` those you want to KEEP.
rights = ChatBannedRights(
    until_date=timedelta(days=7),
    view_messages=None,
    send_messages=None,
    send_media=True,
    send_stickers=True,
    send_gifs=True,
    send_games=True,
    send_inline=True,
    embed_links=True
)

# The above is equivalent to
rights = ChatBannedRights(
    until_date=datetime.now() + timedelta(days=7),
    send_media=True,
    send_stickers=True,
    send_gifs=True,
    send_games=True,
    send_inline=True,
    embed_links=True
)

client(EditBannedRequest(channel, user, rights))
```

You can use a `datetime.datetime` object for `until_date=`, a `datetime.timedelta` or even a Unix timestamp. Note that if you ban someone for less than 30 seconds or for more than 366 days, Telegram will consider the ban to actually last forever. This is officially documented under https://core.telegram.org/bots/api#restrictchatmember.

2.19.8 Kicking a member

Telegram doesn't actually have a request to kick a user from a group. Instead, you need to restrict them so they can't see messages. Any date is enough:

```
from telethon.tl.functions.channels import EditBannedRequest
from telethon.tl.types import ChatBannedRights

client(EditBannedRequest(
    channel, user, ChatBannedRights(
        until_date=None,
        view_messages=True
    )
))
```

2.19.9 Increasing View Count in a Channel

It has been asked quite a few times (really, many), and while I don't understand why so many people ask this, the solution is to use GetMessagesViewsRequest, setting `increment=True`:

```
# Obtain `channel' through dialogs or through client.get_entity() or anyhow.
# Obtain `msg_ids' through `.get_messages()` or anyhow. Must be a list.

client(GetMessagesViewsRequest(
    peer=channel,
    id=msg_ids,
    increment=True
))
```

Note that you can only do this **once or twice a day** per account, running this in a loop will obviously not increase the views forever unless you wait a day between each iteration. If you run it any sooner than that, the views simply won't be increased.

2.20 Users

Note: These examples assume you have read *The Full API*.

> **Contents**
>
> - *Users*
>   - *Retrieving full information*
>   - *Updating your name and/or bio*
>   - *Updating your username*
>   - *Updating your profile photo*

2.20.1 Retrieving full information

If you need to retrieve the bio, biography or about information for a user you should use GetFullUser:

```
from telethon.tl.functions.users import GetFullUserRequest

full = client(GetFullUserRequest(user))
# or even
full = client(GetFullUserRequest('username'))

bio = full.about
```

See UserFull to know what other fields you can access.

2.20.2 Updating your name and/or bio

The first name, last name and bio (about) can all be changed with the same request. Omitted fields won't change after invoking UpdateProfile:

```
from telethon.tl.functions.account import UpdateProfileRequest

client(UpdateProfileRequest(
    about='This is a test from Telethon'
))
```

2.20.3 Updating your username

You need to use account.UpdateUsername:

```
from telethon.tl.functions.account import UpdateUsernameRequest

client(UpdateUsernameRequest('new_username'))
```

2.20.4 Updating your profile photo

The easiest way is to upload a new file and use that as the profile photo through UploadProfilePhoto:

```
from telethon.tl.functions.photos import UploadProfilePhotoRequest

client(UploadProfilePhotoRequest(
    client.upload_file('/path/to/some/file')
)))
```

2.21 Working with messages

Note: These examples assume you have read *The Full API*.

Contents

- *Working with messages*

– *Sending stickers*

2.21.1 Sending stickers

Stickers are nothing else than `files`, and when you successfully retrieve the stickers for a certain sticker set, all you will have are `handles` to these files. Remember, the files Telegram holds on their servers can be referenced through this pair of ID/hash (unique per user), and you need to use this handle when sending a "document" message. This working example will send yourself the very first sticker you have:

```python
# Get all the sticker sets this user has
from telethon.tl.functions.messages import GetAllStickersRequest
sticker_sets = client(GetAllStickersRequest(0))

# Choose a sticker set
from telethon.tl.functions.messages import GetStickerSetRequest
from telethon.tl.types import InputStickerSetID
sticker_set = sticker_sets.sets[0]

# Get the stickers for this sticker set
stickers = client(GetStickerSetRequest(
    stickerset=InputStickerSetID(
        id=sticker_set.id, access_hash=sticker_set.access_hash
    )
))

# Stickers are nothing more than files, so send that
client.send_file('me', stickers.documents[0])
```

2.22 Projects using Telethon

This page lists some **interesting and useful** real world examples showcasing what can be built with the library.

Note: Do you have an interesting project that uses the library or know of any that's not listed here? Feel free to leave a comment at issue 744 so it can be included in the next revision of the documentation!

You can also advertise your bot and its features, in the issue, although it should be a big project which can be useful for others before being included here, so please don't feel offended if it can't be here!

2.22.1 telethon_examples/

telethon_examples / LonamiWebs' site

This documentation is not the only place where you can find useful code snippets using the library. The main repository also has a folder with some cool examples (even a Tkinter GUI!) which you can download, edit and run to learn and play with them.

2.22.2 @TelethonSnippets

@TelethonSnippets

You can find useful short snippets for Telethon here.

2.22.3 telegram-export

telegram-export / expectocode's GitHub

A tool to download Telegram data (users, chats, messages, and media) into a database (and display the saved data).

2.22.4 mautrix-telegram

mautrix-telegram / maunium's site

A Matrix-Telegram hybrid puppeting/relaybot bridge.

2.22.5 TelegramTUI

TelegramTUI / bad-day's GitHub

A Telegram client on your terminal.

2.22.6 tgcloud

tgcloud / tgcloud's site

Opensource Telegram based cloud storage.

2.22.7 tgmount

tgmount / nktknshn's GitHub

Mount Telegram dialogs and channels as a Virtual File System.

2.23 Philosophy

The intention of the library is to have an existing MTProto library existing with hardly any dependencies (indeed, wherever Python is available, you can run this library).

Being written in Python means that performance will be nowhere close to other implementations written in, for instance, Java, C++, Rust, or pretty much any other compiled language. However, the library turns out to actually be pretty decent for common operations such as sending messages, receiving updates, or other scripting. Uploading files may be notably slower, but if you would like to contribute, pull requests are appreciated!

If `libssl` is available on your system, the library will make use of it to speed up some critical parts such as encrypting and decrypting the messages. Files will notably be sent and downloaded faster.

The main focus is to keep everything clean and simple, for everyone to understand how working with MTProto and Telegram works. Don't be afraid to read the source, the code won't bite you! It may prove useful when using the library on your own use cases.

2.24 Test Servers

To run Telethon on a test server, use the following code:

```
client = TelegramClient(None, api_id, api_hash)
client.session.set_dc(dc_id, '149.154.167.40', 80)
```

You can check your `'test ip'` on https://my.telegram.org.

You should set `None` session so to ensure you're generating a new authorization key for it (it would fail if you used a session where you had previously connected to another data center).

Note that port 443 might not work, so you can try with 80 instead.

Once you're connected, you'll likely be asked to either sign in or sign up. Remember anyone can access the phone you choose, so don't store sensitive data here.

Valid phone numbers are `99966XYYYY`, where `X` is the `dc_id` and `YYYY` is any number you want, for example, `1234` in `dc_id = 2` would be `9996621234`. The code sent by Telegram will be `dc_id` repeated five times, in this case, `22222` so we can hardcode that:

```
client = TelegramClient(None, api_id, api_hash)
client.session.set_dc(2, '149.154.167.40', 80)
client.start(
    phone='9996621234', code_callback=lambda: '22222'
)
```

2.25 Project Structure

2.25.1 Main interface

The library itself is under the `telethon/` directory. The `__init__.py` file there exposes the main `TelegramClient`, a class that servers as a nice interface with the most commonly used methods on Telegram such as sending messages, retrieving the message history, handling updates, etc.

The `TelegramClient` inherits from several mixing `Method` classes, since there are so many methods that having them in a single file would make maintenance painful (it was three thousand lines before this separation happened!). It's a "god object", but there is only a way to interact with Telegram really.

The `TelegramBaseClient` is an ABC which will support all of these mixins so they can work together nicely. It doesn't even know how to invoke things because they need to be resolved with user information first (to work with input entities comfortably).

The client makes use of the `network/mtprotosender.py`. The `MTProtoSender` is responsible for connecting, reconnecting, packing, unpacking, sending and receiving items from the network. Basically, the low-level communication with Telegram, and handling MTProto-related functions and types such as `BadSalt`.

The sender makes use of a `Connection` class which knows the format in which outgoing messages should be sent (how to encode their length and their body, if they're further encrypted).

2.25.2 Auto-generated code

The files under `telethon_generator/` are used to generate the code that gets placed under `telethon/tl/`. The parsers take in files in a specific format (such as `.tl` for objects and `.json` for errors) and spit out the generated

classes which represent, as Python classes, the request and types defined in the `.tl` file. It also constructs an index so that they can be imported easily.

Custom documentation can also be generated to easily navigate through the vast amount of items offered by the API.

If you clone the repository, you will have to run `python setup.py gen` in order to generate the code. Installing the library runs the generator too, but the mentioned command will just generate code.

2.26 Coding Style

Basically, make it **readable**, while keeping the style similar to the code of whatever file you're working on.

Also note that not everyone has 4K screens for their primary monitors, so please try to stick to the 80-columns limit. This makes it easy to `git diff` changes from a terminal before committing changes. If the line has to be long, please don't exceed 120 characters.

For the commit messages, please make them *explanatory*. Not only they're helpful to troubleshoot when certain issues could have been introduced, but they're also used to construct the change log once a new version is ready.

If you don't know enough Python, I strongly recommend reading Dive Into Python 3, available online for free. For instance, remember to do `if x is None` or `if x is not None` instead `if x == None`!

2.27 Understanding the Type Language

Telegram's Type Language (also known as TL, found on `.tl` files) is a concise way to define what other programming languages commonly call classes or structs.

Every definition is written as follows for a Telegram object is defined as follows:

```
name#id argument_name:argument_type = CommonType
```

This means that in a single line you know what the `TLObject` name is. You know it's unique ID, and you know what arguments it has. It really isn't that hard to write a generator for generating code to any platform!

The generated code should also be able to *encode* the `TLObject` (let this be a request or a type) into bytes, so they can be sent over the network. This isn't a big deal either, because you know how the `TLObject`'s are made, and how the types should be serialized.

You can either write your own code generator, or use the one this library provides, but please be kind and keep some special mention to this project for helping you out.

This is only a introduction. The `TL` language is not *that* easy. But it's not that hard either. You're free to sniff the `telethon_generator/` files and learn how to parse other more complex lines, such as `flags` (to indicate things that may or may not be written at all) and `vector`'s.

2.28 Tips for Porting the Project

If you're going to use the code on this repository to guide you, please be kind and don't forget to mention it helped you!

You should start by reading the source code on the first release of the project, and start creating a `MTProtoSender`. Once this is made, you should write by hand the code to authenticate on the Telegram's server, which are some steps required to get the key required to talk to them. Save it somewhere! Then, simply mimic, or reinvent other parts of the code, and it will be ready to go within a few days.

Good luck!

2.29 Telegram API in Other Languages

Telethon was made for **Python**, and as far as I know, there is no *exact* port to other languages. However, there *are* other implementations made by awesome people (one needs to be awesome to understand the official Telegram documentation) on several languages (even more Python too), listed below:

2.29.1 C

Possibly the most well-known unofficial open source implementation out there by @vysheng, tgl, and its console client telegram-cli. Latest development has been moved to BitBucket.

C++

The newest (and official) library, written from scratch, is called tdlib and is what the Telegram X uses. You can find more information in the official documentation, published here.

JavaScript

@zerobias is working on telegram-mtproto, a work-in-progress JavaScript library installable via npm.

Kotlin

Kotlogram is a Telegram implementation written in Kotlin (one of the official languages for Android) by @badoualy, currently as a beta– yet working.

PHP

A PHP implementation is also available thanks to @danog and his MadelineProto project, with a very nice online documentation too.

Python

A fairly new (as of the end of 2017) Telegram library written from the ground up in Python by @delivrance and his Pyrogram library. There isn't really a reason to pick it over Telethon and it'd be kinda sad to see you go, but it would be nice to know what you miss from each other library in either one so both can improve.

Rust

Yet another work-in-progress implementation, this time for Rust thanks to @JuanPotato under the fancy name of Vail.

2.30 Changelog (Version History)

This page lists all the available versions of the library, in chronological order. You should read this when upgrading the library to know where your code can break, and where it can take advantage of new goodies!

List of All Versions

2.30.1 Animated Stickers (v1.9)

Published at 2019/07/06

Scheme layer used: 103

With the layer 103, Telethon is now able to send and receive animated stickers! These use the `'application/x-tgsticker'` mime-type and for now, you can access its raw data, which is a gzipped JSON.

Additions

- New `events.Album` to easily receive entire albums!

- New `client.edit_admin()` and `client.edit_permissions()` methods to more easily manage your groups.

- New `pattern=` in `CallbackQuery`.

- New `conversation.cancel_all()` method, to cancel all currently-active conversations in a particular chat.

- New `telethon.utils.encode_waveform` and `telethon.utils.decode_waveform` methods as implemented by Telegram Desktop, which lets you customize how voice notes will render.

- New `ignore_pinned` parameter in `client.iter_dialogs()`.

- New `Message.mark_read()` method.

- You can now use strike-through in markdown with `~~text~~`, and the corresponding HTML tags for strike-through, quotes and underlined text.

- You can now nest entities, as in `**__text__**`.

Bug fixes

- Fixed downloading contacts.

- Fixed `client.iter_dialogs()` missing some under certain circumstances.

- Fixed incredibly slow imports under some systems due to expensive path resolution when searching for `libssl`.

- Fixed captions when sending albums.

- Fixed invalid states in `Conversation`.

- Fixes to some methods in utils regarding extensions.

- Fixed memory cycle in *Forward* which let you do things like the following:

```
original_fwd = message.forward.original_fwd.original_fwd.original_fwd.original_
↪fwd.original_fwd.original_fwd
```

  Hopefully you didn't rely on that in your code.

- Fixed *File.ext* not working on unknown mime-types, despite the file name having the extension.

- Fixed `ids=..., reverse=True` in *client.iter_messages()*.

- Fixed *Draft* not being aware of the entity.

- Added missing re-exports in `telethon.sync`.

Enhancements

- Improved *conversation.cancel()* behaviour. Now you can use it from anywhere.

- The `progress_callback` in *client.download_media()* now lets you use `async def`.

- Improved documentation and the online method reference at https://tl.telethon.dev.

2.30.2 Documentation Overhaul (v1.8)

Published at 2019/05/30

Scheme layer used: 100

The documentation has been completely reworked from the ground up, with awesome new quick references such as *Client Reference* to help you quickly find what you need!

Raw methods also warn you when a friendly variant is available, so that you don't accidentally make your life harder than it has to be.

In addition, all methods in the client now are fully annotated with type hints! More work needs to be done, but this should already help a lot when using Telethon from any IDEs.

You may have noticed that the patch versions between `v1.7.2` to `v1.7.7` have not been documented. This is because patch versions should only contain bug fixes, no new features or breaking changes. This hasn't been the case in the past, but from now on, the library will try to adhere more strictly to the Semantic Versioning principles.

If you ever want to look at those bug fixes, please use the appropriated `git` command, such as `git shortlog v1.7.1...v1.7.4`, but in general, they probably just fixed your issue.

With that out of the way, let's look at the full change set:

Breaking Changes

- The layer changed, so take note if you use the raw API, as it's usual.

- The way photos are downloaded changed during the layer update of the previous version, and fixing that bug as a breaking change in itself. *client.download_media()* now offers a different way to deal with thumbnails.

Additions

- New *Message.file* property! Now you can trivially access *message.file.id* to get the file ID of some media, or even `print(message.file.name)`.

- Archiving dialogs with *Dialog.archive()* or *client.edit_folder()* is now possible.

- New cleaned-up method to stream downloads with *client.iter_download()*, which offers a lot of flexibility, such as arbitrary offsets for efficient seeking.

- *Dialog.delete()* has existed for a while, and now *client.delete_dialog()* exists too so you can easily leave chats or delete dialogs without fetching all dialogs.

- Some people or chats have a lot of profile photos. You can now iterate over all of them with the new *client.iter_profile_photos()* method.

- You can now annoy everyone with the new *Message.pin(notify=True)*! The client has its own variant too, called *client.pin_message()*.

Bug fixes

- Correctly catch and raise all RPC errors.

- Downloading stripped photos wouldn't work correctly.

- Under some systems, `libssl` would fail to load earlier than expected, causing the library to fail when being imported.

- *conv.get_response()* after ID 0 wasn't allowed when it should.

- *InlineBuilder* only worked with local files, but files from anywhere are supported.

- Accessing the text property from a raw-API call to fetch Message would fail (any any other property that needed the client).

- Database is now upgraded if the version was lower, not different. From now on, this should help with upgrades and downgrades slightly.

- Fixed saving `pts` and session-related stuff.

- Disconnection should not raise any errors.

- Invite links of the form `tg://join?invite=` now work.

- *client.iter_participants(search=...)* now works on private chats again.

- Iterating over messages in reverse with a date as offset wouldn't work.

- The conversation would behave weirdly when a timeout occurred.

Enhancements

- `telethon` now re-export all the goodies that you commonly need when using the library, so e.g. `from telethon import Button` will now work.

- `telethon.sync` now re-exports everything from `telethon`, so that you can trivially import from just one place everything that you need.

- More attempts at reducing CPU usage after automatically fetching missing entities on events. This isn't a big deal, even if it sounds like one.

- Hexadecimal invite links are now supported. You didn't need them, but they will now work.

Internal Changes

- Deterministic code generation. This is good for `diff`.

- On Python 3.7 and above, we properly close the connection.

- A lot of micro-optimization.

- Fixes to bugs introduced while making this release.

- Custom commands on `setup.py` are nicer to use.

2.30.3 Fix-up for Photo Downloads (v1.7.1)

Published at 2019/04/24

Telegram changed the way thumbnails (which includes photos) are downloaded, so you can no longer use a PhotoSize alone to download a particular thumbnail size (this is a **breaking change**).

Instead, you will have to specify the new `thumb` parameter in `client.download_media()` to download a particular thumbnail size. This addition enables you to easily download thumbnails from documents, something you couldn't do easily before.

2.30.4 Easier Events (v1.7)

Published at 2019/04/22

Scheme layer used: 98

If you have been using Telethon for a while, you probably know how annoying the "Could not find the input entity for. . ." error can be. In this new version, the library will try harder to find the input entity for you!

That is, instead of doing:

```python
@client.on(events.NewMessage)
async def handler(event):
    await client.download_profile_photo(await event.get_input_sender())
    # ...... needs await, it's a method ^^^^^                    ^^
```

You can now do:

```python
@client.on(events.NewMessage)
async def handler(event):
    await client.download_profile_photo(event.input_sender)
    # ...... no await, it's a property! ^
    # It's also 12 characters shorter :)
```

And even the following will hopefully work:

```python
@client.on(events.NewMessage)
async def handler(event):
    await client.download_profile_photo(event.sender_id)
```

A lot of people use IDs thinking this is the right way of doing it. Ideally, you would always use `input_*`, not `sender` or `sender_id` (and the same applies to chats). But, with this change, IDs will work just the same as `input_*` inside events.

This feature still needs some more testing, so please do open an issue if you find strange behaviour.

Breaking Changes

- The layer changed, and a lot of things did too. If you are using raw API, you should be careful with this. In addition, some attributes weren't of type `datetime` when they should be, which has been fixed.

- Due to the layer change, you can no longer download photos with just their PhotoSize. Version 1.7.1 introduces a new way to download thumbnails to work around this issue.

- `client.disconnect()` is now asynchronous again. This means you need to `await` it. You don't need to worry about this if you were using `with client` or `client.run_until_disconnected`. This should prevent the "pending task was destroyed" errors.

Additions

- New in-memory cache for input entities. This should mean a lot less of disk look-ups.

- New `client.action` method to easily indicate that you are doing some chat action:

```
async with client.action(chat, 'typing'):
    await asyncio.sleep(2)  # type for 2 seconds
    await client.send_message(chat, 'Hello world! I type slow ^^')
```

You can also easily use this for sending files, playing games, etc.

New bugs

- Downloading photos is broken. This is fixed in v1.7.1.

Bug fixes

- Fix sending photos from streams/bytes.

- Fix unhandled error when sending requests that were too big.

- Fix edits that arrive too early on conversations.

- Fix `client.edit_message()` when trying to edit a file.

- Fix method calls on the objects returned by `client.iter_dialogs()`.

- Attempt at fixing `client.iter_dialogs()` missing many dialogs.

- `offset_date` in `client.iter_messages()` was being ignored in some cases. This has been worked around.

- Fix `callback_query.edit()`.

- Fix `CallbackQuery(func=...)` was being ignored.

- Fix `UserUpdate` not working for "typing" (and uploading file, etc.) status.

- Fix library was not expecting `IOError` from PySocks.

- Fix library was raising a generic `ConnectionError` and not the one that actually occurred.

- Fix the `blacklist_chats` parameter in `MessageRead` not working as intended.

- Fix `client.download_media(contact)`.

- Fix mime type when sending `mp3` files.

- Fix forcibly getting the sender or chat from events would not always return all their information.

- Fix sending albums with *client.send_file()* was not returning the sent messages.

- Fix forwarding albums with *client.forward_messages()*.

- Some fixes regarding filtering updates from chats.

- Attempt at preventing duplicated updates.

- Prevent double auto-reconnect.

Enhancements

- Some improvements related to proxy connections.

- Several updates and improvements to the documentation, such as optional dependencies now being properly listed.

- You can now forward messages from different chats directly with *client.forward_messages*.

2.30.5 Tidying up Internals (v1.6)

Published at 2019/02/27

Scheme layer used: 95

First things first, sorry for updating the layer in the previous patch version. That should only be done between major versions ideally, but due to how Telegram works, it's done between minor versions. However raw API has and will always be considered "unsafe", this meaning that you should always use the convenience client methods instead. These methods don't cover the full API yet, so pull requests are welcome.

Breaking Changes

- The layer update, of course. This didn't really need a mention here.

- You can no longer pass a `batch_size` when iterating over messages. No other method exposed this parameter, and it was only meant for testing purposes. Instead, it's now a private constant.

- `client.iter_*` methods no longer have a `_total` parameter which was supposed to be private anyway. Instead, they return a new generator object which has a `.total` attribute:

```
it = client.iter_messages(chat)
for i, message in enumerate(it, start=1):
    percentage = i / it.total
    print('{:.2%} {}'.format(percentage, message.text))
```

Additions

- You can now pass `phone` and `phone_code_hash` in *client.sign_up*, although you probably don't need that.

- Thanks to the overhaul of all `client.iter_*` methods, you can now do:

```
for message in reversed(client.iter_messages('me')):
    print(message.text)
```

Bug fixes

- Fix *telethon.utils.resolve_bot_file_id*, which wasn't working after the layer update (so you couldn't send some files by bot file IDs).

- Fix sending albums as bot file IDs (due to image detection improvements).

- Fix *takeout()* failing when they need to download media from other DCs.

- Fix repeatedly calling *conversation.get_response()* when many messages arrived at once (i.e. when several of them were forwarded).

- Fixed connecting with *ConnectionTcpObfuscated*.

- Fix *client.get_peer_id('me')*.

- Fix warning of "missing sqlite3" when in reality it just had wrong tables.

- Fix a strange error when using too many IDs in *client.delete_messages()*.

- Fix *client.send_file* with the result of *client.upload_file*.

- When answering inline results, their order was not being preserved.

- Fix *events.ChatAction* detecting user leaves as if they were kicked.

Enhancements

- Cleared up some parts of the documentation.

- Improved some auto-casts to make life easier.

- Improved image detection. Now you can easily send `bytes` and streams of images as photos, unless you force document.

- Sending images as photos that are too large will now be resized before uploading, reducing the time it takes to upload them and also avoiding errors when the image was too large (as long as `pillow` is installed). The images will remain unchanged if you send it as a document.

- Treat `errors.RpcMcgetFailError` as a temporary server error to automatically retry shortly. This works around most issues.

Internal changes

- New common way to deal with retries (`retry_range`).

- Cleaned up the takeout client.

- Completely overhauled asynchronous generators.

2.30.6 Layer Update (v1.5.5)

Published at 2019/01/14

Scheme layer used: 93

There isn't an entry for v1.5.4 because it contained only one hot-fix regarding loggers. This update is slightly bigger so it deserves mention.

Additions

- New `supports_streaming` parameter in *`client.send_file`*.

Bug fixes

- Dealing with mimetypes should cause less issues in systems like Windows.

- Potentially fix alternative session storages that had issues with dates.

Enhancements

- Saner timeout defaults for conversations.

- `Path`-like files are now supported for thumbnails.

- Added new hot-keys to the online documentation at https://tl.telethon.dev/ such as / to search. Press ? to view them all.

2.30.7 Bug Fixes (v1.5.3)

Published at 2019/01/14

Several bug fixes and some quality of life enhancements.

Breaking Changes

- *`message.edit`* now respects the previous message buttons or link preview being hidden. If you want to toggle them you need to explicitly set them. This is generally the desired behaviour, but may cause some bots to have buttons when they shouldn't.

Additions

- You can now "hide_via" when clicking on results from *`client.inline_query`* to @bing and @gif.

- You can now further configure the logger Telethon uses to suit your needs.

Bug fixes

- Fixes for ReadTheDocs to correctly build the documentation.

- Fix UserEmpty not being expected when getting the input variant.

- The message object returned when sending a message with buttons wouldn't always contain the ReplyMarkup.

- Setting email when configuring 2FA wasn't properly supported.

- `utils.resolve_bot_file_id` now works again for photos.

Enhancements

- Chat and channel participants can now be used as peers.

- Reworked README and examples at https://github.com/LonamiWebs/Telethon/tree/master/telethon_examples

---

2.30.8 Takeout Sessions (v1.5.2)

Published at 2019/01/05

You can now easily start takeout sessions (also known as data export sessions) through `client.takeout()`. Some of the requests will have lower flood limits when done through the takeout session.

Bug fixes

- The new `AdminLogEvent` had a bug that made it unusable.
- `client.iter_dialogs()` will now locally check for the offset date, since Telegram ignores it.
- Answering inline queries with media no works properly. You can now use the library to create inline bots and send stickers through them!

2.30.9 object.to_json() (v1.5.1)

Published at 2019/01/03

The library already had a way to easily convert the objects the API returned into dictionaries through `object.to_dict()`, but some of the fields are dates or `bytes` which JSON can't serialize directly.

For convenience, a new `object.to_json()` has been added which will by default format both of those problematic types into something sensible.

Additions

- New `client.iter_admin_log()` method.

Bug fixes

- `client.is_connected()` would be wrong when the initial connection failed.
- Fixed `UnicodeDecodeError` when accessing the text of messages with malformed offsets in their entities.
- Fixed `client.get_input_entity()` for integer IDs that the client has not seen before.

Enhancements

- You can now configure the reply markup when using `Button` as a bot.
- More properties for `Message` to make accessing media convenient.
- Downloading to `file=bytes` will now return a `bytes` object with the downloaded media.

2.30.10 Polls with the Latest Layer (v1.5)

Published at 2018/12/25

Scheme layer used: 91

This version doesn't really bring many new features, but rather focuses on updating the code base to support the latest available Telegram layer, 91. This layer brings polls, and you can create and manage them through Telethon!

Breaking Changes

- The layer change from 82 to 91 changed a lot of things in the raw API, so be aware that if you rely on raw API calls, you may need to update your code, in particular **if you work with files**. They have a new `file_reference` parameter that you must provide.

Additions

- New *`client.is_bot()`* method.

Bug fixes

- Markdown and HTML parsing now behave correctly with leading whitespace.

- HTTP connection should now work correctly again.

- Using `caption=None` would raise an error instead of setting no caption.

- `KeyError` is now handled properly when forwarding messages.

- *`button.click()`* now works as expected for KeyboardButtonGame.

Enhancements

- Some improvements to the search in the full API and generated examples.

- Using entities with `access_hash = 0` will now work in more cases.

Internal changes

- Some changes to the documentation and code generation.

- 2FA code was updated to work under the latest layer.

2.30.11 Error Descriptions in CSV files (v1.4.3)

Published at 2018/12/04

While this may seem like a minor thing, it's a big usability improvement.

Anyone who wants to update the documentation for known errors, or whether some methods can be used as a bot, user or both, can now be easily edited. Everyone is encouraged to help document this better!

Bug fixes

- `TimeoutError` was not handled during automatic reconnects.

- Getting messages by ID using InputMessageReplyTo could fail.

- Fixed *`message.get_reply_message`* as a bot when a user replied to a different bot.

- Accessing some document properties in a *`Message`* would fail.

Enhancements

- Accessing *events.ChatAction* properties such as input users may now work in more cases.

Internal changes

- Error descriptions and information about methods is now loaded from a CSV file instead of being part of several messy JSON files.

2.30.12 Bug Fixes (v1.4.2)

Published at 2018/11/24

This version also includes the v1.4.1 hot-fix, which was a single quick fix and didn't really deserve an entry in the changelog.

Bug fixes

- Authorization key wouldn't be saved correctly, requiring re-login.
- Conversations with custom events failed to be cancelled.
- Fixed `telethon.sync` when using other threads.
- Fix markdown/HTML parser from failing with leading/trailing whitespace.
- Fix accessing `chat_action_event.input_user` property.
- Potentially improved handling unexpected disconnections.

Enhancements

- Better default behaviour for *client.send_read_acknowledge*.
- Clarified some points in the documentation.
- Clearer errors for `utils.get_peer*`.

2.30.13 Connection Overhaul (v1.4)

Published at 2018/11/03

Yet again, a lot of work has been put into reworking the low level connection classes. This means `asyncio.open_connection` is now used correctly and the errors it can produce are handled properly. The separation between packing, encrypting and network is now abstracted away properly, so reasoning about the code is easier, making it more maintainable.

As a user, you shouldn't worry about this, other than being aware that quite a few changes were made in the insides of the library and you should report any issues that you encounter with this version if any.

Breaking Changes

- The threaded version of the library will no longer be maintained, primarily because it never was properly maintained anyway. If you have old code, stick with old versions of the library, such as `0.19.1.6`.

- Timeouts no longer accept `timedelta`. Simply use seconds.

- The `callback` parameter from `telethon.tl.custom.button.Button.inline()` was removed, since it had always been a bad idea. Adding the callback there meant a lot of extra work for every message sent, and only registering it after the first message was sent! Instead, use `telethon.events.callbackquery.CallbackQuery`.

Additions

- New `dialog.delete()` method.
- New `conversation.cancel()` method.
- New `retry_delay` delay for the client to be used on auto-reconnection.

Bug fixes

- Fixed `Conversation.wait_event()`.
- Fixed replying with photos/documents on inline results.
- `client.is_user_authorized()` now works correctly after `client.log_out()`.
- `dialog.is_group` now works for ChatForbidden.
- Not using `async with` when needed is now a proper error.
- `events.CallbackQuery` with string regex was not working properly.
- `client.get_entity('me')` now works again.
- Empty codes when signing in are no longer valid.
- Fixed file cache for in-memory sessions.

Enhancements

- Support `next_offset` in `inline_query.answer()`.
- Support `<a href="tg://user?id=123">` mentions in HTML parse mode.
- New auto-casts for InputDocument and InputChatPhoto.
- Conversations are now exclusive per-chat by default.
- The request that caused a RPC error is now shown in the error message.
- New full API examples in the generated documentation.
- Fixed some broken links in the documentation.
- `client.disconnect()` is now synchronous, but you can still `await` it for consistency or compatibility.

2.30.14 Event Templates (v1.3)

Published at 2018/09/22

If you have worked with Flask templates, you will love this update, since it gives you the same features but even more conveniently:

```python
# handlers/welcome.py
from telethon import events


@events.register(events.NewMessage('(?i)hello'))
async def handler(event):
    client = event.client
    await event.respond('Hi!')
    await client.send_message('me', 'Sent hello to someone')
```

This will *register* the `handler` callback to handle new message events. Note that you didn't add this to any client yet, and this is the key point: you don't need a client to define handlers! You can add it later:

```python
# main.py
from telethon import TelegramClient
import handlers.welcome

with TelegramClient(...) as client:
    # This line adds the handler we defined before for new messages
    client.add_event_handler(handlers.welcome.handler)
    client.run_until_disconnected()
```

This should help you to split your big code base into a more modular design.

Breaking Changes

- `.sender` is the `.chat` when the message is sent in a broadcast channel. This makes sense, because the sender of the message was the channel itself, but you now must take into consideration that it may be either a User or Channel instead of being `None`.

Additions

- New `MultiError` class when invoking many requests at once through `client([requests])`.
- New custom `func=` on all events. These will receive the entire event, and a good usage example is `func=lambda e: e.is_private`.
- New `.web_preview` field on messages. The `.photo` and `.document` will also return the media in the web preview if any, for convenience.
- Callback queries now have a `.chat` in most circumstances.

Bug fixes

- Running code with `python3 -O` would remove critical code from asserts.
- Fix some rare ghost disconnections after reconnecting.
- Fix strange behavior for `send_message(chat, Message, reply_to=foo)`.
- The `loop=` argument was being pretty much ignored.

- Fix `MemorySession` file caching.

- The logic for getting entities from their username is now correct.

- Fixes for sending stickers from `.webp` files in Windows, again.

- Fix disconnection without being logged in.

- Retrieving media from messages would fail.

- Getting some messages by ID on private chats.

Enhancements

- *`iter_participants`* will now use its `search=` as a symbol set when `aggressive=True`, so you can do `client.get_participants(group, aggressive=True, search='')`.

- The `StringSession` supports custom encoding.

- Callbacks for *`telethon.client.auth.AuthMethods.start`* can be `async`.

Internal changes

- Cherry-picked a commit to use `asyncio.open_connection` in the lowest level of the library. Do open issues if this causes trouble, but it should otherwise improve performance and reliability.

- Building and resolving events overhaul.

2.30.15 Conversations, String Sessions and More (v1.2)

Published at 2018/08/14

This is a big release! Quite a few things have been added to the library, such as the new *`Conversation`*. This makes it trivial to get tokens from [@BotFather](#):

```python
from telethon.tl import types

with client.conversation('BotFather') as conv:
    conv.send_message('/mybots')
    message = conv.get_response()
    message.click(0)
    message = conv.get_edit()
    message.click(0)
    message = conv.get_edit()
    for _, token in message.get_entities_text(types.MessageEntityCode):
        print(token)
```

In addition to that, you can now easily load and export session files without creating any on-disk file thanks to the `StringSession`:

```python
from telethon.sessions import StringSession
string = StringSession.save(client.session)
```

Check out *Session Files* for more details.

For those who aren't able to install `cryptg`, the support for `libssl` has been added back. While interfacing `libssl` is not as fast, the speed when downloading and sending files should really be noticeably faster.

While those are the biggest things, there are still more things to be excited about.

Additions

- The mentioned method to start a new *client.conversation*.

- Implemented global search through *client.iter_messages* with `None` entity.

- New *client.inline_query* method to perform inline queries.

- Bot-API-style `file_id` can now be used to send files and download media. You can also access *telethon. utils.resolve_bot_file_id* and *telethon.utils.pack_bot_file_id* to resolve and create these file IDs yourself. Note that each user has its own ID for each file so you can't use a bot's `file_id` with your user, except stickers.

- New *telethon.utils.get_peer*, useful when you expect a Peer.

Bug fixes

- UTC timezone for *telethon.events.userupdate.UserUpdate*.

- Bug with certain input parameters when iterating messages.

- RPC errors without parent requests caused a crash, and better logging.

- `incoming = outgoing = True` was not working properly.

- Getting a message's ID was not working.

- File attributes not being inferred for `open()`'ed files.

- Use `MemorySession` if `sqlite3` is not installed by default.

- Self-user would not be saved to the session file after signing in.

- *client.catch_up()* seems to be functional again.

Enhancements

- Updated documentation.

- Invite links will now use cache, so using them as entities is cheaper.

- You can reuse message buttons to send new messages with those buttons.

- `.to_dict()` will now work even on invalid `TLObject`'s.

2.30.16 Better Custom Message (v1.1.1)

Published at 2018/07/23

The *custom.Message* class has been rewritten in a cleaner way and overall feels less hacky in the library. This should perform better than the previous way in which it was patched.

The release is primarily intended to test this big change, but also fixes **Python 3.5.2 compatibility** which was broken due to a trailing comma.

Bug fixes

- Using `functools.partial` on event handlers broke updates if they had uncaught exceptions.

- A bug under some session files where the sender would export authorization for the same data center, which is unsupported.

- Some logical bugs in the custom message class.

2.30.17 Bot Friendly (v1.1)

Published at 2018/07/21

Two new event handlers to ease creating normal bots with the library, namely *events.InlineQuery* and *events.CallbackQuery* for handling @InlineBot queries or reacting to a button click. For this second option, there is an even better way:

```python
from telethon.tl.custom import Button

async def callback(event):
    await event.edit('Thank you!')

bot.send_message(chat, 'Hello!',
                 buttons=Button.inline('Click me', callback))
```

You can directly pass the callback when creating the button.

This is fine for small bots but it will add the callback every time you send a message, so you probably should do this instead once you are done testing:

```python
markup = bot.build_reply_markup(Button.inline('Click me', callback))
bot.send_message(chat, 'Hello!', buttons=markup)
```

And yes, you can create more complex button layouts with lists:

```python
from telethon import events

global phone = ''

@bot.on(events.CallbackQuery)
async def handler(event):
    global phone
    if event.data == b'<':
        phone = phone[:-1]
    else:
        phone += event.data.decode('utf-8')

    await event.answer('Phone is now {}'.format(phone))

markup = bot.build_reply_markup([
    [Button.inline('1'), Button.inline('2'), Button.inline('3')],
    [Button.inline('4'), Button.inline('5'), Button.inline('6')],
    [Button.inline('7'), Button.inline('8'), Button.inline('9')],
    [Button.inline('+'), Button.inline('0'), Button.inline('<')],
])
bot.send_message(chat, 'Enter a phone', buttons=markup)
```

(Yes, there are better ways to do this). Now for the rest of things:

Additions

- New *custom.Button* class to help you create inline (or normal) reply keyboards. You must sign in as a bot to use the buttons= parameters.

- New events usable if you sign in as a bot: *events.InlineQuery* and *events.CallbackQuery*.

- New silent parameter when sending messages, usable in broadcast channels.

- Documentation now has an entire section dedicate to how to use the client's friendly methods at *(removed broken link)*.

Bug fixes

- Empty except are no longer used which means sending a keyboard interrupt should now work properly.

- The pts of incoming updates could be None.

- UTC timezone information is properly set for read datetime.

- Some infinite recursion bugs in the custom message class.

- Updates was being dispatched to raw handlers when it shouldn't.

- Using proxies and HTTPS connection mode may now work properly.

- Less flood waits when downloading media from different data centers, and the library will now detect them even before sending requests.

Enhancements

- Interactive sign in now supports signing in with a bot token.

- timedelta is now supported where a date is expected, which means you can e.g. ban someone for timedelta(minutes=5).

- Events are only built once and reused many times, which should save quite a few CPU cycles if you have a lot of the same type.

- You can now click inline buttons directly if you know their data.

Internal changes

- When downloading media, the right sender is directly used without previously triggering migrate errors.

- Code reusing for getting the chat and the sender, which easily enables this feature for new types.

2.30.18 New HTTP(S) Connection Mode (v1.0.4)

Published at 2018/07/09

This release implements the HTTP connection mode to the library, which means certain proxies that only allow HTTP connections should now work properly. You can use it doing the following, like any other mode:

```
from telethon import TelegramClient, sync
from telethon.network import ConnectionHttp

client = TelegramClient(..., connection=ConnectionHttp)
with client:
    client.send_message('me', 'Hi!')
```

Additions

- `add_mark=` is now back on `utils.get_input_peer` and also on *client.get_input_entity()*.

- New *client.get_peer_id* convenience for `utils.get_peer_id(await client.get_input_entity(peer))`.

Bug fixes

- If several `TLMessage` in a `MessageContainer` exceeds 1MB, it will no longer be automatically turned into one. This basically means that e.g. uploading 10 file parts at once will work properly again.

- Documentation fixes and some missing `await`.

- Revert named argument for *client.forward_messages*

Enhancements

- New auto-casts to [InputNotifyPeer](#) and `chat_id`.

Internal changes

- Outgoing `TLMessage` are now pre-packed so if there's an error when serializing the raw requests, the library will no longer swallow it. This also means re-sending packets doesn't need to re-pack their bytes.

2.30.19 Iterate Messages in Reverse (v1.0.3)

Published at 2018/07/04

Scheme layer used: 82

Mostly bug fixes, but now there is a new parameter on *client.iter_messages* to support reversing the order in which messages are returned.

Additions

- The mentioned `reverse` parameter when iterating over messages.

- A new `sequential_updates` parameter when creating the client for updates to be processed sequentially. This is useful when you need to make sure that all updates are processed in order, such as a script that only forwards incoming messages somewhere else.

Bug fixes

- Count was always `None` for *message.button_count*.

- Some fixes when disconnecting upon dropping the client.

- Support for Python 3.4 in the sync version, and fix media download.

- Some issues with events when accessing the input chat or their media.

- Hachoir wouldn't automatically close the file after reading its metadata.

- Signing in required a named `code=` parameter, but usage without a name was really widespread so it has been reverted.

2.30.20 Bug Fixes (v1.0.2)

Published at 2018/06/28

Updated some asserts and parallel downloads, as well as some fixes for sync.

2.30.21 Bug Fixes (v1.0.1)

Published at 2018/06/27

And as usual, every major release has a few bugs that make the library unusable! This quick update should fix those, namely:

Bug fixes

- *client.start()* was completely broken due to a last-time change requiring named arguments everywhere.

- Since the rewrite, if your system clock was wrong, the connection would get stuck in an infinite "bad message" loop of responses from Telegram.

- Accessing the buttons of a custom message wouldn't work in channels, which lead to fix a completely different bug regarding starting bots.

- Disconnecting could complain if the magic `telethon.sync` was imported.

- Successful automatic reconnections now ask Telegram to send updates to us once again as soon as the library is ready to listen for them.

2.30.22 Synchronous magic (v1.0)

Published at 2018/06/27

Important: If you come from Telethon pre-1.0 you **really** want to read *Compatibility and Convenience* to port your scripts to the new version.

The library has been around for well over a year. A lot of improvements have been made, a lot of user complaints have been fixed, and a lot of user desires have been implemented. It's time to consider the public API as stable, and remove some of the old methods that were around until now for compatibility reasons. But there's one more surprise!

There is a new magic `telethon.sync` module to let you use **all** the methods in the *TelegramClient* (and the types returned from its functions) in a synchronous way, while using `asyncio` behind the scenes! This means you're now able to do both of the following:

```python
import asyncio

async def main():
    await client.send_message('me', 'Hello!')

asyncio.get_event_loop().run_until_complete(main())

# ...can be rewritten as:

from telethon import sync
client.send_message('me', 'Hello!')
```

Both ways can coexist (you need to `await` if the loop is running).

You can also use the magic `sync` module in your own classes, and call `sync.syncify(cls)` to convert all their `async def` into magic variants.

Breaking Changes

- `message.get_fwd_sender` is now in *message.forward*.
- `client.idle` is now *client.run_until_disconnected()*
- `client.add_update_handler` is now *client.add_event_handler*
- `client.remove_update_handler` is now *client.remove_event_handler*
- `client.list_update_handlers` is now *client.list_event_handlers*
- `client.get_message_history` is now *client.get_messages*
- `client.send_voice_note` is now *client.send_file* with `is_voice=True`.
- `client.invoke()` is now `client(...)`.
- `report_errors` has been removed since it's currently not used, and `flood_sleep_threshold` is now part of the client.
- The `update_workers` and `spawn_read_thread` arguments are gone. Simply remove them from your code when you create the client.
- Methods with a lot of arguments can no longer be used without specifying their argument. Instead you need to use named arguments. This improves readability and not needing to learn the order of the arguments, which can also change.

Additions

- *client.send_file* now accepts external `http://` and `https://` URLs.
- You can use the *TelegramClient* inside of `with` blocks, which will *client.start()* and *disconnect()* the client for you:

```python
from telethon import TelegramClient, sync

with TelegramClient(name, api_id, api_hash) as client:
    client.send_message('me', 'Hello!')
```

Convenience at its maximum! You can even chain the `.start()` method since it returns the instance of the client:

```
with TelegramClient(name, api_id, api_hash).start(bot_token=token) as bot:
    bot.send_message(chat, 'Hello!')
```

Bug fixes

- There were some `@property async def` left, and some `await property`.
- "User joined" event was being treated as "User was invited".
- SQLite's cursor should not be closed properly after usage.
- `await` the updates task upon disconnection.
- Some bug in Python 3.5.2's `asyncio` causing 100% CPU load if you forgot to call *client. disconnect()*. The method is called for you on object destruction, but you still should disconnect manually or use a `with` block.
- Some fixes regarding disconnecting on client deletion and properly saving the authorization key.
- Passing a class to *message.get_entities_text* now works properly.
- Iterating messages from a specific user in private messages now works.

Enhancements

- Both *client.start()* and *client.run_until_disconnected()* can be ran in both a synchronous way (without starting the loop manually) or from an `async def` where they need to have an `await`.

2.30.23 Core Rewrite in asyncio (v1.0-rc1)

Published at 2018/06/24

Scheme layer used: 81

This version is a major overhaul of the library internals. The core has been rewritten, cleaned up and refactored to fix some oddities that have been growing inside the library.

This means that the code is easier to understand and reason about, including the code flow such as conditions, exceptions, where to reconnect, how the library should behave, and separating different retry types such as disconnections or call fails, but it also means that **some things will necessarily break** in this version.

All requests that touch the network are now methods and need to have their `await` (or be ran until their completion).

Also, the library finally has the simple logo it deserved: a carefully hand-written `.svg` file representing a T following Python's colours.

Breaking Changes

- If you relied on internals like the `MtProtoSender` and the `TelegramBareClient`, both are gone. They are now `MTProtoSender` and *TelegramBaseClient* and they behave differently.
- Underscores have been renamed from filenames. This means `telethon.errors.rpc_error_list` won't work, but you should have been using `telethon.errors` all this time instead.

- *client.connect* no longer returns `True` on success. Instead, you should `except` the possible `ConnectionError` and act accordingly. This makes it easier to not ignore the error.

- You can no longer set `retries=n` when calling a request manually. The limit works differently now, and it's done on a per-client basis.

- Accessing `.sender`, `.chat` and similar may *not* work in events anymore, since previously they could access the network. The new rule is that properties are not allowed to make API calls. You should use `.get_sender()`, `.get_chat()` instead while using events. You can safely access properties if you get messages through *client.get_messages()* or other methods in the client.

- The above point means `reply_message` is now *.get_reply_message()*, and `fwd_from_entity` is now `get_fwd_sender()`. Also `forward` was gone in the previous version, and you should be using `fwd_from` instead.

Additions

- Telegram's Terms Of Service are now accepted when creating a new account. This can possibly help avoid bans. This has no effect for accounts that were created before.

- The method reference now shows which methods can be used if you sign in with a `bot_token`.

- There's a new *client.disconnected* future which you can wait on. When a disconnection occurs, you will now, instead letting it happen in the background.

- More configurable retries parameters, such as auto-reconnection, retries when connecting, and retries when sending a request.

- You can filter *events.NewMessage* by sender ID, and also whether they are forwards or not.

- New `ignore_migrated` parameter for *client.iter_dialogs*.

Bug fixes

- Several fixes to *telethon.events.newmessage.NewMessage*.

- Removed named `length` argument in `to_bytes` for PyPy.

- Raw events failed due to not having `._set_client`.

- *message.get_entities_text* properly supports filtering, even if there are no message entities.

- *message.click* works better.

- The server started sending DraftMessageEmpty which the library didn't handle correctly when getting dialogs.

- The "correct" chat is now always returned from returned messages.

- `to_id` was not validated when retrieving messages by their IDs.

- `'__'` is no longer considered valid in usernames.

- The `fd` is removed from the reader upon closing the socket. This should be noticeable in Windows.

- MessageEmpty is now handled when searching messages.

- Fixed a rare infinite loop bug in *client.iter_dialogs* for some people.

- Fixed `TypeError` when there is no `.sender`.

Enhancements

- You can now delete over 100 messages at once with *client.delete_messages*.

- Signing in now accounts for AuthRestartError itself, and also handles PasswordHashInvalidError.

- __all__ is now defined, so from telethon import * imports sane defaults (client, events and utils). This is however discouraged and should be used only in quick scripts.

- pathlib.Path is now supported for downloading and uploading media.

- Messages you send to yourself are now considered outgoing, unless they are forwarded.

- The documentation has been updated with a brand new asyncio crash course to encourage you use it. You can still use the threaded version if you want though.

- .name property is now properly supported when sending and downloading files.

- Custom parse_mode, which can now be set per-client, support MessageEntityMentionName so you can return those now.

- The session file is saved less often, which could result in a noticeable speed-up when working with a lot of incoming updates.

Internal changes

- The flow for sending a request is as follows: the TelegramClient creates a MTProtoSender with a Connection, and the sender starts send and receive loops. Sending a request means enqueueing it in the sender, which will eventually pack and encrypt it with its ConnectionState instead of using the entire Session instance. When the data is packed, it will be sent over the Connection and ultimately over the TcpClient.

- Reconnection occurs at the MTProtoSender level, and receiving responses follows a similar process, but now asyncio.Future is used for the results which are no longer part of all TLObject, instead are part of the TLMessage which simplifies things.

- Objects can no longer be content_related and instead subclass TLRequest, making the separation of concerns easier.

- The TelegramClient has been split into several mixin classes to avoid having a 3,000-lines-long file with all the methods.

- More special cases in the MTProtoSender have been cleaned up, and also some attributes from the Session which didn't really belong there since they weren't being saved.

- The telethon_generator/ can now convert .tl files into .json, mostly as a proof of concept, but it might be useful for other people.

2.30.24 Custom Message class (v0.19.1)

Published at 2018/06/03

Scheme layer used: 80

This update brings a new *telethon.tl.custom.message.Message* object!

All the methods in the telethon.telegram_client.TelegramClient that used to return a Message will now return this object instead, which means you can do things like the following:

```
msg = client.send_message(chat, 'Hello!')
msg.edit('Hello there!')
msg.reply('Good day!')
print(msg.sender)
```

Refer to its documentation to see all you can do, again, click *telethon.tl.custom.message.Message* to go
to its page.

Breaking Changes

- The `telethon.network.connection.common.Connection` class is now an ABC, and the old
  `ConnectionMode` is now gone. Use a specific connection (like *telethon.network.connection.*
  tcpabridged.ConnectionTcpAbridged) instead.

Additions

- You can get messages by their ID with `telethon.telegram_client.TelegramClient.`
  `get_messages`'s `ids` parameter:

```
message = client.get_messages(chats, ids=123)  # Single message
message_list = client.get_messages(chats, ids=[777, 778])  # Multiple
```

- More convenience properties for *telethon.tl.custom.dialog.Dialog*.

- New default `telethon.telegram_client.TelegramClient.parse_mode`.

- You can edit the media of messages that already have some media.

- New dark theme in the online `tl` reference, check it out at https://tl.telethon.dev/.

Bug fixes

- Some IDs start with `1000` and these would be wrongly treated as channels.

- Some short usernames like `@vote` were being ignored.

- `telethon.telegram_client.TelegramClient.iter_messages`'s `from_user` was failing if
  no filter had been set.

- `telethon.telegram_client.TelegramClient.iter_messages`'s `min_id/max_id` was be-
  ing ignored by Telegram. This is now worked around.

- `telethon.telegram_client.TelegramClient.catch_up` would fail with empty states.

- *telethon.events.newmessage.NewMessage* supports `incoming=False` to indicate
  `outgoing=True`.

Enhancements

- You can now send multiple requests at once while preserving the order:

```
from telethon.tl.functions.messages import SendMessageRequest
client([SendMessageRequest(chat, 'Hello 1!'),
        SendMessageRequest(chat, 'Hello 2!')], ordered=True)
```

Internal changes

- `without rowid` is not used in SQLite anymore.

- Unboxed serialization would fail.

- Different default limit for `iter_messages` and `get_messages`.

- Some clean-up in the `telethon_generator/` package.

2.30.25 Catching up on Updates (v0.19)

Published at 2018/05/07

Scheme layer used: 76

This update prepares the library for catching up with updates with the new `telethon.telegram_client.TelegramClient.catch_up` method. This feature needs more testing, but for now it will let you "catch up" on some old updates that occurred while the library was offline, and brings some new features and bug fixes.

Additions

- Add `search`, `filter` and `from_user` parameters to `telethon.telegram_client.TelegramClient.iter_messages`.

- `telethon.telegram_client.TelegramClient.download_file` now supports a `None` path to return the file in memory and return its `bytes`.

- Events now have a `.original_update` field.

Bug fixes

- Fixed a race condition when receiving items from the network.

- A disconnection is made when "retries reached 0". This hasn't been tested but it might fix the bug.

- `reply_to` would not override Message object's reply value.

- Add missing caption when sending Message with media.

Enhancements

- Retry automatically on `RpcCallFailError`. This error happened a lot when iterating over many messages, and retrying often fixes it.

- Faster `telethon.telegram_client.TelegramClient.iter_messages` by sleeping only as much as needed.

- `telethon.telegram_client.TelegramClient.edit_message` now supports omitting the entity if you pass a Message.

- `telethon.events.raw.Raw` can now be filtered by type.

Internal changes

- The library now distinguishes between MTProto and API schemas.
- State is now persisted to the session file.
- Connection won't retry forever.
- Fixed some errors and cleaned up the generation of code.
- Fixed typos and enhanced some documentation in general.
- Add auto-cast for InputMessage and InputLocation.

2.30.26 Pickle-able objects (v0.18.3)

Published at 2018/04/15

Now you can use Python's `pickle` module to serialize `RPCError` and any other `TLObject` thanks to **@vegeta1k95**! A fix that was fairly simple, but still might be useful for many people.

As a side note, the documentation at https://tl.telethon.dev now lists known `RPCError` for all requests, so you know what to expect. This required a major rewrite, but it was well worth it!

Breaking changes

- `telethon.telegram_client.TelegramClient.forward_messages` now returns a single item instead of a list if the input was also a single item.

Additions

- New *telethon.events.messageread.MessageRead* event, to find out when and who read which messages as soon as it happens.
- Now you can access `.chat_id` on all events and `.sender_id` on some.

Bug fixes

- Possibly fix some bug regarding lost `GzipPacked` requests.
- The library now uses the "real" layer 75, hopefully.
- Fixed `.entities` name collision on updates by making it private.
- `AUTH_KEY_DUPLICATED` is handled automatically on connection.
- Markdown parser's offset uses `match.start()` to allow custom regex.
- Some filter types (as a type) were not supported by `telethon.telegram_client.TelegramClient.iter_participants`.
- `telethon.telegram_client.TelegramClient.remove_event_handler` works.
- `telethon.telegram_client.TelegramClient.start` works on all terminals.
- InputPeerSelf case was missing from `telethon.telegram_client.TelegramClient.get_input_entity`.

Enhancements

- The `parse_mode` for messages now accepts a callable.
- `telethon.telegram_client.TelegramClient.download_media` accepts web previews.
- *`telethon.tl.custom.dialog.Dialog`* instances can now be casted into InputPeer.
- Better logging when reading packages "breaks".
- Better and more powerful `setup.py gen` command.

Internal changes

- The library won't call `.get_dialogs()` on entity not found. Instead, it will `raise ValueError()` so you can properly `except` it.
- Several new examples and updated documentation.
- `py:obj` is the default Sphinx's role which simplifies `.rst` files.
- `setup.py` now makes use of `python_requires`.
- Events now live in separate files.
- Other minor changes.

2.30.27 Several bug fixes (v0.18.2)

Published at 2018/03/27

Just a few bug fixes before they become too many.

Additions

- Getting an entity by its positive ID should be enough, regardless of their type (whether it's an `User`, a `Chat` or a `Channel`). Although wrapping them inside a `Peer` is still recommended, it's not necessary.
- New `client.edit_2fa` function to change your Two Factor Authentication settings.
- `.stringify()` and string representation for custom `Dialog/Draft`.

Bug fixes

- Some bug regarding `.get_input_peer`.
- `events.ChatAction` wasn't picking up all the pins.
- `force_document=True` was being ignored for albums.
- Now you're able to send `Photo` and `Document` as files.
- Wrong access to a member on chat forbidden error for `.get_participants`. An empty list is returned instead.
- `me/self` check for `.get[_input]_entity` has been moved up so if someone has "me" or "self" as their name they won't be retrieved.

2.30.28 Iterator methods (v0.18.1)

Published at 2018/03/17

All the `.get_` methods in the `TelegramClient` now have a `.iter_` counterpart, so you can do operations while retrieving items from them. For instance, you can `client.iter_dialogs()` and `break` once you find what you're looking for instead fetching them all at once.

Another big thing, you can get entities by just their positive ID. This may cause some collisions (although it's very unlikely), and you can (should) still be explicit about the type you want. However, it's a lot more convenient and less confusing.

Breaking changes

- The library only offers the default `SQLiteSession` again. See *Session Files* for more on how to use a different storage from now on.

Additions

- Events now override `__str__` and implement `.stringify()`, just like every other `TLObject` does.
- `events.ChatAction` now has `respond()`, `reply()` and `delete()` for the message that triggered it.
- `client.iter_participants()` (and its `client.get_participants()` counterpart) now expose the `filter` argument, and the returned users also expose the `.participant` they are.
- You can now use `client.remove_event_handler()` and `client.list_event_handlers()` similar how you could with normal updates.
- New properties on `events.NewMessage`, like `.video_note` and `.gif` to access only specific types of documents.
- The `Draft` class now exposes `.text` and `.raw_text`, as well as a new `Draft.send()` to send it.

Bug fixes

- `MessageEdited` was ignoring `NewMessage` constructor arguments.
- Fixes for `Event.delete_messages` which wouldn't handle `MessageService`.
- Bot API style IDs not working on `client.get_input_entity()`.
- `client.download_media()` didn't support `PhotoSize`.

Enhancements

- Less RPC are made when accessing the `.sender` and `.chat` of some events (mostly those that occur in a channel).
- You can send albums larger than 10 items (they will be sliced for you), as well as mixing normal files with photos.
- `TLObject` now have Python type hints.

Internal changes

- Several documentation corrections.
- `client.get_dialogs()` is only called once again when an entity is not found to avoid flood waits.

2.30.29 Sessions overhaul (v0.18)

Published at 2018/03/04

Scheme layer used: 75

The `Session`'s have been revisited thanks to the work of **@tulir** and they now use an ABC so you can easily implement your own!

The default will still be a `SQLiteSession`, but you might want to use the new `AlchemySessionContainer` if you need. Refer to the section of the documentation on *Session Files* for more.

Breaking changes

- `events.MessageChanged` doesn't exist anymore. Use the new `events.MessageEdited` and `events.MessageDeleted` instead.

Additions

- The mentioned addition of new session types.
- You can omit the event type on `client.add_event_handler` to use `Raw`.
- You can `raise StopPropagation` of events if you added several of them.
- `.get_participants()` can now get up to 90,000 members from groups with 100,000 if when `aggressive=True`, "bypassing" Telegram's limit.
- You now can access `NewMessage.Event.pattern_match`.
- Multiple captions are now supported when sending albums.
- `client.send_message()` has an optional `file=` parameter, so you can do `events.reply(file='/path/to/photo.jpg')` and similar.
- Added `.input_` versions to `events.ChatAction`.
- You can now access the public `.client` property on `events`.
- New `client.forward_messages`, with its own wrapper on `events`, called `event.forward_to(...)`.

Bug fixes

- Silly bug regarding `client.get_me(input_peer=True)`.
- `client.send_voice_note()` was missing some parameters.
- `client.send_file()` plays better with streams now.
- Incoming messages from bots weren't working with whitelists.

- Markdown's URL regex was not accepting newlines.
- Better attempt at joining background update threads.
- Use the right peer type when a marked integer ID is provided.

Internal changes

- Resolving `events.Raw` is now a no-op.
- Logging calls in the `TcpClient` to spot errors.
- `events` resolution is postponed until you are successfully connected, so you can attach them before starting the client.
- When an entity is not found, it is searched in *all* dialogs. This might not always be desirable but it's more comfortable for legitimate uses.
- Some non-persisting properties from the `Session` have been moved out.

2.30.30 Further easing library usage (v0.17.4)

Published at 2018/02/24

Some new things and patches that already deserved their own release.

Additions

- New `pattern` argument to `NewMessage` to easily filter messages.
- New `.get_participants()` convenience method to get members from chats.
- `.send_message()` now accepts a `Message` as the `message` parameter.
- You can now `.get_entity()` through exact name match instead username.
- Raise `ProxyConnectionError` instead looping forever so you can `except` it on your own code and behave accordingly.

Bug fixes

- `.parse_username` would fail with `www.` or a trailing slash.
- `events.MessageChanged` would fail with `UpdateDeleteMessages`.
- You can now send `b'byte strings'` directly as files again.
- `.send_file()` was not respecting the original captions when passing another message (or media) as the file.
- Downloading media from a different data center would always log a warning for the first time.

Internal changes

- Use `req_pq_multi` instead `req_pq` when generating `auth_key`.
- You can use `.get_me(input_peer=True)` if all you need is your self ID.
- New addition to the interactive client example to show peer information.

- Avoid special casing `InputPeerSelf` on some `NewMessage` events, so you can always safely rely on `.sender` to get the right ID.

2.30.31 New small convenience functions (v0.17.3)

Published at 2018/02/18

More bug fixes and a few others addition to make events easier to use.

Additions

- Use `hachoir` to extract video and audio metadata before upload.
- New `.add_event_handler`, `.add_update_handler` now deprecated.

Bug fixes

- `bot_token` wouldn't work on `.start()`, and changes to `password` (now it will ask you for it if you don't provide it, as docstring hinted).
- `.edit_message()` was ignoring the formatting (e.g. markdown).
- Added missing case to the `NewMessage` event for normal groups.
- Accessing the `.text` of the `NewMessage` event was failing due to a bug with the markdown unparser.

Internal changes

- `libssl` is no longer an optional dependency. Use `cryptg` instead, which you can find on https://github.com/Lonami/cryptg.

2.30.32 New small convenience functions (v0.17.2)

Published at 2018/02/15

Primarily bug fixing and a few welcomed additions.

Additions

- New convenience `.edit_message()` method on the `TelegramClient`.
- New `.edit()` and `.delete()` shorthands on the `NewMessage` event.
- Default to markdown parsing when sending and editing messages.
- Support for inline mentions when sending and editing messages. They work like inline urls (e.g. `[text](@username)`) and also support the Bot-API style (see here).

Bug fixes

- Periodically send `GetStateRequest` automatically to keep the server sending updates even if you're not invoking any request yourself.

- HTML parsing was failing due to not handling surrogates properly.

- `.sign_up` was not accepting `int` codes.

- Whitelisting more than one chat on `events` wasn't working.

- Video files are sent as a video by default unless `force_document`.

Internal changes

- More `logging` calls to help spot some bugs in the future.

- Some more logic to retrieve input entities on events.

- Clarified a few parts of the documentation.

2.30.33 Updates as Events (v0.17.1)

Published at 2018/02/09

Of course there was more work to be done regarding updates, and it's here! The library comes with a new `events` module (which you will often import as `from telethon import TelegramClient, events`). This are pretty much all the additions that come with this version change, but they are a nice addition. Refer to *(removed broken link)* to get started with events.

2.30.34 Trust the Server with Updates (v0.17)

Published at 2018/02/03

The library trusts the server with updates again. The library will *not* check for duplicates anymore, and when the server kicks us, it will run `GetStateRequest` so the server starts sending updates again (something it wouldn't do unless you invoked something, it seems). But this update also brings a few more changes!

Additions

- `TLObject`'s override `__eq__` and `__ne__`, so you can compare them.

- Added some missing cases on `.get_input_entity()` and peer functions.

- `obj.to_dict()` now has a `'_'` key with the type used.

- `.start()` can also sign up now.

- More parameters for `.get_message_history()`.

- Updated list of RPC errors.

- HTML parsing thanks to **@tulir**! It can be used similar to markdown: `client.send_message(..., parse_mode='html')`.

Enhancements

- `client.send_file()` now accepts `Message`'s and `MessageMedia`'s as the `file` parameter.

- Some documentation updates and fixed to clarify certain things.

- New exact match feature on https://tl.telethon.dev.

- Return as early as possible from `.get_input_entity()` and similar, to avoid penalizing you for doing this right.

Bug fixes

- `.download_media()` wouldn't accept a `Document` as parameter.

- The SQLite is now closed properly on disconnection.

- IPv6 addresses shouldn't use square braces.

- Fix regarding `.log_out()`.

- The time offset wasn't being used (so having wrong system time would cause the library not to work at all).

2.30.35 New `.resolve()` method (v0.16.2)

Published at 2018/01/19

The `TLObject`'s (instances returned by the API and `Request`'s) have now acquired a new `.resolve()` method. While this should be used by the library alone (when invoking a request), it means that you can now use `Peer` types or even usernames where a `InputPeer` is required. The object now has access to the `client`, so that it can fetch the right type if needed, or access the session database. Furthermore, you can reuse requests that need "autocast" (e.g. you put `User` but `InputPeer` was needed), since `.resolve()` is called when invoking. Before, it was only done on object construction.

Additions

- Album support. Just pass a list, tuple or any iterable to `.send_file()`.

Enhancements

- `.start()` asks for your phone only if required.

- Better file cache. All files under 10MB, once uploaded, should never be needed to be re-uploaded again, as the sent media is cached to the session.

Bug fixes

- `setup.py` now calls `gen_tl` when installing the library if needed.

Internal changes

- The mentioned `.resolve()` to perform "autocast", more powerful.

- Upload and download methods are no longer part of `TelegramBareClient.`

- Reuse `.on_response()`, `.__str__` and `.stringify()`. Only override `.on_response()` if necessary (small amount of cases).

- Reduced "autocast" overhead as much as possible. You shouldn't be penalized if you've provided the right type.

2.30.36 MtProto 2.0 (v0.16.1)

Published at 2018/01/11

Scheme layer used: 74

The library is now using MtProto 2.0! This shouldn't really affect you as an end user, but at least it means the library will be ready by the time MtProto 1.0 is deprecated.

Additions

- New `.start()` method, to make the library avoid boilerplate code.

- `.send_file` accepts a new optional `thumbnail` parameter, and returns the `Message` with the sent file.

Bug fixes

- The library uses again only a single connection. Less updates are be dropped now, and the performance is even better than using temporary connections.

- `without rowid` will only be used on the `*.session` if supported.

- Phone code hash is associated with phone, so you can change your mind when calling `.sign_in()`.

Internal changes

- File cache now relies on the hash of the file uploaded instead its path, and is now persistent in the `*.session` file. Report any bugs on this!

- Clearer error when invoking without being connected.

- Markdown parser doesn't work on bytes anymore (which makes it cleaner).

2.30.37 Sessions as sqlite databases (v0.16)

Published at 2017/12/28

In the beginning, session files used to be pickle. This proved to be bad as soon as one wanted to add more fields. For this reason, they were migrated to use JSON instead. But this proved to be bad as soon as one wanted to save things like entities (usernames, their ID and hash), so now it properly uses sqlite3, which has been well tested, to save the session files! Calling `.get_input_entity` using a `username` no longer will need to fetch it first, so it's really 0 calls again. Calling `.get_entity` will always fetch the most up to date version.

Furthermore, nearly everything has been documented, thus preparing the library for Read the Docs (although there are a few things missing I'd like to polish first), and the logging are now better placed.

Breaking changes

- `.get_dialogs()` now returns a **single list** instead a tuple consisting of a **custom class** that should make everything easier to work with.

- `.get_message_history()` also returns a **single list** instead a tuple, with the `Message` instances modified to make them more convenient.

Both lists have a `.total` attribute so you can still know how many dialogs/messages are in total.

Additions

- The mentioned use of `sqlite3` for the session file.

- `.get_entity()` now supports lists too, and it will make as little API calls as possible if you feed it `InputPeer` types. Usernames will always be resolved, since they may have changed.

- `.set_proxy()` method, to avoid having to create a new `TelegramClient`.

- More `date` types supported to represent a date parameter.

Bug fixes

- Empty strings weren't working when they were a flag parameter (e.g., setting no last name).

- Fix invalid assertion regarding flag parameters as well.

- Avoid joining the background thread on disconnect, as it would be `None` due to a race condition.

- Correctly handle `None` dates when downloading media.

- `.download_profile_photo` was failing for some channels.

- `.download_media` wasn't handling `Photo`.

Internal changes

- `date` was being serialized as local date, but that was wrong.

- `date` was being represented as a `float` instead of an `int`.

- `.tl` parser wasn't stripping inline comments.

- Removed some redundant checks on `update_state.py`.

- Use a synchronized queue instead a hand crafted version.

- Use signed integers consistently (e.g. `salt`).

- Always read the corresponding `TLObject` from API responses, except for some special cases still.

- A few more `except` low level to correctly wrap errors.

- More accurate exception types.

- `invokeWithLayer(initConnection(X))` now wraps every first request after `.connect()`.

As always, report if you have issues with some of the changes!

2.30.38 IPv6 support (v0.15.5)

Published at 2017/11/16

Scheme layer used: 73

It's here, it has come! The library now **supports IPv6**! Just pass `use_ipv6=True` when creating a `TelegramClient`. Note that I could *not* test this feature because my machine doesn't have IPv6 setup. If you know IPv6 works in your machine but the library doesn't, please refer to #425.

Additions

- IPv6 support.

- New method to extract the text surrounded by `MessageEntity`'s, in the `extensions.markdown` module.

Enhancements

- Markdown parsing is Done Right.

- Reconnection on failed invoke. Should avoid "number of retries reached 0" (#270).

- Some missing autocast to `Input*` types.

- The library uses the `NullHandler` for `logging` as it should have always done.

- `TcpClient.is_connected()` is now more reliable.

Bug fixes

- Getting an entity using their phone wasn't actually working.

- Full entities aren't saved unless they have an `access_hash`, to avoid some `None` errors.

- `.get_message_history` was failing when retrieving items that had messages forwarded from a channel.

2.30.39 General enhancements (v0.15.4)

Published at 2017/11/04

Scheme layer used: 72

This update brings a few general enhancements that are enough to deserve a new release, with a new feature: beta **markdown-like parsing** for `.send_message()`!

Additions

- `.send_message()` supports `parse_mode='md'` for **Markdown**! It works in a similar fashion to the official clients (defaults to double underscore/asterisk, like `**this**`). Please report any issues with emojies or enhancements for the parser!

- New `.idle()` method so your main thread can do useful job (listen for updates).

- Add missing `.to_dict()`, `__str__` and `.stringify()` for `TLMessage` and `MessageContainer`.

Bug fixes

- The list of known peers could end "corrupted" and have users with `access_hash=None`, resulting in `struct` error for it not being an integer. You shouldn't encounter this issue anymore.

- The warning for "added update handler but no workers set" wasn't actually working.

- `.get_input_peer` was ignoring a case for `InputPeerSelf`.

- There used to be an exception when logging exceptions (whoops) on update handlers.

- "Downloading contacts" would produce strange output if they had semicolons (`;`) in their name.

- Fix some cyclic imports and installing dependencies from the `git` repository.

- Code generation was using f-strings, which are only supported on Python 3.6.

Internal changes

- The `auth_key` generation has been moved from `.connect()` to `.invoke()`. There were some issues were `.connect()` failed and the `auth_key` was `None` so this will ensure to have a valid `auth_key` when needed, even if `BrokenAuthKeyError` is raised.

- Support for higher limits on `.get_history()` and `.get_dialogs()`.

- Much faster integer factorization when generating the required `auth_key`. Thanks @delivrance for making me notice this, and for the pull request.

2.30.40 Bug fixes with updates (v0.15.3)

Published at 2017/10/20

Hopefully a very ungrateful bug has been removed. When you used to invoke some request through update handlers, it could potentially enter an infinite loop. This has been mitigated and it's now safe to invoke things again! A lot of updates were being dropped (all those gzipped), and this has been fixed too.

More bug fixes include a correct parsing of certain TLObjects thanks to @stek29, and some wrong calls that would cause the library to crash thanks to @andr-04, and the `ReadThread` not re-starting if you were already authorized.

Internally, the `.to_bytes()` function has been replaced with `__bytes__` so now you can do `bytes(tlobject)`.

2.30.41 Bug fixes and new small features (v0.15.2)

Published at 2017/10/14

This release primarily focuses on a few bug fixes and enhancements. Although more stuff may have broken along the way.

Enhancements

- You will be warned if you call `.add_update_handler` with no `update_workers`.

- New customizable threshold value on the session to determine when to automatically sleep on flood waits. See `client.session.flood_sleep_threshold`.

- New `.get_drafts()` method with a custom `Draft` class by @JosXa.

- Join all threads when calling `.disconnect()`, to assert no dangling thread is left alive.
- Larger chunk when downloading files should result in faster downloads.
- You can use a callable key for the `EntityDatabase`, so it can be any filter you need.

Bug fixes

- `.get_input_entity` was failing for IDs and other cases, also making more requests than it should.
- Use `basename` instead `abspath` when sending a file. You can now also override the attributes.
- `EntityDatabase.__delitem__` wasn't working.
- `.send_message()` was failing with channels.
- `.get_dialogs(limit=None)` should now return all the dialogs correctly.
- Temporary fix for abusive duplicated updates.

Internal changes

- MsgsAck is now sent in a container rather than its own request.
- `.get_input_photo` is now used in the generated code.
- `.process_entities` was being called from more places than only `__call__`.
- `MtProtoSender` now relies more on the generated code to read responses.

2.30.42 Custom Entity Database (v0.15.1)

Published at 2017/10/05

The main feature of this release is that Telethon now has a custom database for all the entities you encounter, instead depending on `@lru_cache` on the `.get_entity()` method.

The `EntityDatabase` will, by default, **cache** all the users, chats and channels you find in memory for as long as the program is running. The session will, by default, save all key-value pairs of the entity identifiers and their hashes (since Telegram may send an ID that it thinks you already know about, we need to save this information).

You can **prevent** the `EntityDatabase` from saving users by setting `client.session.entities.enabled = False`, and prevent the `Session` from saving input entities at all by setting `client.session.save_entities = False`. You can also clear the cache for a certain user through `client.session.entities.clear_cache(entity=None)`, which will clear all if no entity is given.

Additions

- New method to `.delete_messages()`.
- New `ChannelPrivateError` class.

Enhancements

- `.sign_in` accepts phones as integers.
- Changing the IP to which you connect to is as simple as `client.session.server_address = 'ip'`, since now the server address is always queried from the session.

Bug fixes

- `.get_dialogs()` doesn't fail on Windows anymore, and returns the right amount of dialogs.

- `GeneralProxyError` should be passed to the main thread again, so that you can handle it.

2.30.43 Updates Overhaul Update (v0.15)

Published at 2017/10/01

After hundreds of lines changed on a major refactor, *it's finally here*. It's the **Updates Overhaul Update**; let's get right into it!

Breaking changes

- `.create_new_connection()` is gone for good. No need to deal with this manually since new connections are now handled on demand by the library itself.

Enhancements

- You can **invoke** requests from **update handlers**. And **any other thread**. A new temporary will be made, so that you can be sending even several requests at the same time!

- **Several worker threads** for your updates! By default, `None` will spawn. I recommend you to work with `update_workers=4` to get started, these will be polling constantly for updates.

- You can also change the number of workers at any given time.

- The library can now run **in a single thread** again, if you don't need to spawn any at all. Simply set `spawn_read_thread=False` when creating the `TelegramClient`!

- You can specify `limit=None` on `.get_dialogs()` to get **all** of them[1].

- **Updates are expanded**, so you don't need to check if the update has `.updates` or an inner `.update` anymore.

- All `InputPeer` entities are **saved in the session** file, but you can disable this by setting `save_entities=False`.

- New `.get_input_entity` method, which makes use of the above feature. You **should use this** when a request needs a `InputPeer`, rather than the whole entity (although both work).

- Assert that either all or None dependent-flag parameters are set before sending the request.

- Phone numbers can have dashes, spaces, or parenthesis. They'll be removed before making the request.

- You can override the phone and its hash on `.sign_in()`, if you're creating a new `TelegramClient` on two different places.

Bug fixes

- `.log_out()` was consuming all retries. It should work just fine now.

- The session would fail to load if the `auth_key` had been removed manually.

- `Updates.check_error` was popping wrong side, although it's been completely removed.

- `ServerError`'s will be **ignored**, and the request will immediately be retried.

- Cross-thread safety when saving the session file.

- Some things changed on a matter of when to reconnect, so please report any bugs!

Internal changes

- `TelegramClient` is now only an abstraction over the `TelegramBareClient`, which can only do basic things, such as invoking requests, working with files, etc. If you don't need any of the abstractions the `TelegramClient`, you can now use the `TelegramBareClient` in a much more comfortable way.

- `MtProtoSender` is not thread-safe, but it doesn't need to be since a new connection will be spawned when needed.

- New connections used to be cached and then reused. Now only their sessions are saved, as temporary connections are spawned only when needed.

- Added more RPC errors to the list.

[1]: Broken due to a condition which should had been the opposite (sigh), fixed 4 commits ahead on https://github.com/LonamiWebs/Telethon/commit/62ea77cbeac7c42bfac85aa8766a1b5b35e3a76c.

That's pretty much it, although there's more work to be done to make the overall experience of working with updates *even better*. Stay tuned!

2.30.44 Serialization bug fixes (v0.14.2)

Published at 2017/09/29

Bug fixes

- **Important**, related to the serialization. Every object or request that had to serialize a `True/False` type was always being serialized as `false`!

- Another bug that didn't allow you to leave as `None` flag parameters that needed a list has been fixed.

Internal changes

- Other internal changes include a somewhat more readable `.to_bytes()` function and pre-computing the flag instead using bit shifting. The `TLObject.constructor_id` has been renamed to `TLObject.CONSTRUCTOR_ID`, and `.subclass_of_id` is also uppercase now.

2.30.45 Farewell, BinaryWriter (v0.14.1)

Published at 2017/09/28

Version `v0.14` had started working on the new `.to_bytes()` method to dump the `BinaryWriter` and its usage on the `.on_send()` when serializing TLObjects, and this release finally removes it. The speed up when serializing things to bytes should now be over twice as fast wherever it's needed.

Bug fixes

- This version is again compatible with Python 3.x versions **below 3.5** (there was a method call that was Python 3.5 and above).

Internal changes

- Using proper classes (including the generated code) for generating authorization keys and to write out `TLMessage`'s.

2.30.46 Several requests at once and upload compression (v0.14)

Published at 2017/09/27

New major release, since I've decided that these two features are big enough:

Additions

- Requests larger than 512 bytes will be **compressed through gzip**, and if the result is smaller, this will be uploaded instead.
- You can now send **multiple requests at once**, they're simply `*var_args` on the `.invoke()`. Note that the server doesn't guarantee the order in which they'll be executed!

Internally, another important change. The `.on_send` function on the `TLObjects` is **gone**, and now there's a new `.to_bytes()`. From my tests, this has always been over twice as fast serializing objects, although more replacements need to be done, so please report any issues.

Enhancements

- Implemented `.get_input_media` helper methods. Now you can even use another message as input media!

Bug fixes

- Downloading media from CDNs wasn't working (wrong access to a parameter).
- Correct type hinting.
- Added a tiny sleep when trying to perform automatic reconnection.
- Error reporting is done in the background, and has a shorter timeout.
- `setup.py` used to fail with wrongly generated code.

2.30.47 Quick fix-up (v0.13.6)

Published at 2017/09/23

Before getting any further, here's a quick fix-up with things that should have been on `v0.13.5` but were missed. Specifically, the **timeout when receiving** a request will now work properly.

Some other additions are a tiny fix when **handling updates**, which was ignoring some of them, nicer `__str__` and `.stringify()` methods for the `TLObject`'s, and not stopping the `ReadThread` if you try invoking something there (now it simply returns `None`).

2.30.48 Attempts at more stability (v0.13.5)

Published at 2017/09/23

Yet another update to fix some bugs and increase the stability of the library, or, at least, that was the attempt!

This release should really **improve the experience with the background thread** that the library starts to read things from the network as soon as it can, but I can't spot every use case, so please report any bug (and as always, minimal reproducible use cases will help a lot).

Bug fixes

- `setup.py` was failing on Python < 3.5 due to some imports.

- Duplicated updates should now be ignored.

- `.send_message` would crash in some cases, due to having a typo using the wrong object.

- `"socket is None"` when calling `.connect()` should not happen anymore.

- `BrokenPipeError` was still being raised due to an incorrect order on the `try/except` block.

Enhancements

- **Type hinting** for all the generated `Request`'s and `TLObjects`! IDEs like PyCharm will benefit from this.

- `ProxyConnectionError` should properly be passed to the main thread for you to handle.

- The background thread will only be started after you're authorized on Telegram (i.e. logged in), and several other attempts at polishing the experience with this thread.

- The `Connection` instance is only created once now, and reused later.

- Calling `.connect()` should have a better behavior now (like actually *trying* to connect even if we seemingly were connected already).

- `.reconnect()` behavior has been changed to also be more consistent by making the assumption that we'll only reconnect if the server has disconnected us, and is now private.

Internal changes

- `TLObject.__repr__` doesn't show the original TL definition anymore, it was a lot of clutter. If you have any complaints open an issue and we can discuss it.

- Internally, the `'+'` from the phone number is now stripped, since it shouldn't be included.

- Spotted a new place where `BrokenAuthKeyError` would be raised, and it now is raised there.

2.30.49 More bug fixes and enhancements (v0.13.4)

Published at 2017/09/18

Additions

- `TelegramClient` now exposes a `.is_connected()` method.
- Initial authorization on a new data center will retry up to 5 times by default.
- Errors that couldn't be handled on the background thread will be raised on the next call to `.invoke()` or `updates.poll()`.

Bug fixes

- Now you should be able to sign in even if you have `process_updates=True` and no previous session.
- Some errors and methods are documented a bit clearer.
- `.send_message()` could randomly fail, as the returned type was not expected.
- `TimeoutError` is now ignored, since the request will be retried up to 5 times by default.
- "-404" errors (`BrokenAuthKeyError`'s) are now detected when first connecting to a new data center.
- `BufferError` is handled more gracefully, in the same way as `InvalidCheckSumError`'s.
- Attempt at fixing some "NoneType has no attribute..." errors (with the `.sender`).

Internal changes

- Calling `GetConfigRequest` is now made less often.
- The `initial_query` parameter from `.connect()` is gone, as it's not needed anymore.
- Renamed `all_tlobjects.layer` to `all_tlobjects.LAYER` (since it's a constant).
- The message from `BufferError` is now more useful.

2.30.50 Bug fixes and enhancements (v0.13.3)

Published at 2017/09/14

Bug fixes

- **Reconnection** used to fail because it tried invoking things from the `ReadThread`.
- Inferring **random ids** for `ForwardMessagesRequest` wasn't working.
- Downloading media from **CDNs** failed due to having forgotten to remove a single line.
- `TcpClient.close()` now has a **"threading.Lock"**, so `NoneType has no close()` should not happen.
- New **workaround** for `msg seqno too low/high`. Also, both `Session.id/seq` are not saved anymore.

Enhancements

- **Request will be retried** up to 5 times by default rather than failing on the first attempt.

- `InvalidChecksumError`'s are now **ignored** by the library.

- `TelegramClient.get_entity()` is now **public**, and uses the `@lru_cache()` decorator.

- New method to **''.send_voice_note()''**'s.

- Methods to send message and media now support a **''reply_to'' parameter**.

- `.send_message()` now returns the **full message** which was just sent.

2.30.51 New way to work with updates (v0.13.2)

Published at 2017/09/08

This update brings a new way to work with updates, and it's begging for your **feedback**, or better names or ways to do what you can do now.

Please refer to the wiki/Usage Modes for an in-depth description on how to work with updates now. Notice that you cannot invoke requests from within handlers anymore, only the `v.0.13.1` patch allowed you to do so.

Bug fixes

- Periodic pings are back.

- The username regex mentioned on `UsernameInvalidError` was invalid, but it has now been fixed.

- Sending a message to a phone number was failing because the type used for a request had changed on layer 71.

- CDN downloads weren't working properly, and now a few patches have been applied to ensure more reliability, although I couldn't personally test this, so again, report any feedback.

2.30.52 Invoke other requests from within update callbacks (v0.13.1)

Published at 2017/09/04

> **Warning:** This update brings some big changes to the update system, so please read it if you work with them!

A silly "bug" which hadn't been spotted has now been fixed. Now you can invoke other requests from within your update callbacks. However **this is not advised**. You should post these updates to some other thread, and let that thread do the job instead. Invoking a request from within a callback will mean that, while this request is being invoked, no other things will be read.

Internally, the generated code now resides under a *lot* less files, simply for the sake of avoiding so many unnecessary files. The generated code is not meant to be read by anyone, simply to do its job.

Unused attributes have been removed from the `TLObject` class too, and `.sign_up()` returns the user that just logged in in a similar way to `.sign_in()` now.

2.30.53 Connection modes (v0.13)

Published at 2017/09/04

Scheme layer used: 71

The purpose of this release is to denote a big change, now you can connect to Telegram through different **connection modes**. Also, a **second thread** will *always* be started when you connect a `TelegramClient`, despite whether you'll be handling updates or ignoring them, whose sole purpose is to constantly read from the network.

The reason for this change is as simple as *"reading and writing shouldn't be related"*. Even when you're simply ignoring updates, this way, once you send a request you will only need to read the result for the request. Whatever Telegram sent before has already been read and outside the buffer.

Additions

- The mentioned different connection modes, and a new thread.

- You can modify the `Session` attributes through the `TelegramClient` constructor (using `**kwargs`).

- `RPCError`'s now belong to some request you've made, which makes more sense.

- `get_input_*` now handles `None` (default) parameters more gracefully (it used to crash).

Enhancements

- The low-level socket doesn't use a handcrafted timeout anymore, which should benefit by avoiding the arbitrary `sleep(0.1)` that there used to be.

- `TelegramClient.sign_in` will call `.send_code_request` if no `code` was provided.

Deprecation

- `.sign_up` does *not* take a `phone` argument anymore. Change this or you will be using `phone` as `code`, and it will fail! The definition looks like `def sign_up(self, code, first_name, last_name='')`.

- The old `JsonSession` finally replaces the original `Session` (which used pickle). If you were overriding any of these, you should only worry about overriding `Session` now.

2.30.54 Added verification for CDN file (v0.12.2)

Published at 2017/08/28

Since the Content Distributed Network (CDN) is not handled by Telegram itself, the owners may tamper these files. Telegram sends their sha256 sum for clients to implement this additional verification step, which now the library has. If any CDN has altered the file you're trying to download, `CdnFileTamperedError` will be raised to let you know.

Besides this. `TLObject.stringify()` was showing bytes as lists (now fixed) and RPC errors are reported by default:

> In an attempt to help everyone who works with the Telegram API, Telethon will by default report all Remote Procedure Call errors to PWRTelegram, a public database anyone can query, made by Daniil. All the information sent is a GET request with the error code, error message and method used.

Note: If you still would like to opt out, simply set `client.session.report_errors = False` to disable this feature. However Daniil would really thank you if you helped him (and everyone) by keeping it on!

2.30.55 CDN support (v0.12.1)

Published at 2017/08/24

The biggest news for this update are that downloading media from CDN's (you'll often encounter this when working with popular channels) now **works**.

Bug fixes

- The method used to download documents crashed because two lines were swapped.

- Determining the right path when downloading any file was very weird, now it's been enhanced.

- The `.sign_in()` method didn't support integer values for the code! Now it does again.

Some important internal changes are that the old way to deal with RSA public keys now uses a different module instead the old strange hand-crafted version.

Hope the new, super simple `README.rst` encourages people to use Telethon and make it better with either suggestions, or pull request. Pull requests are *super* appreciated, but showing some support by leaving a star also feels nice .

2.30.56 Newbie friendly update (v0.12)

Published at 2017/08/22

Scheme layer used: 70

This update is overall an attempt to make Telethon a bit more user friendly, along with some other stability enhancements, although it brings quite a few changes.

Breaking changes

- The `TelegramClient` methods `.send_photo_file()`, `.send_document_file()` and `.send_media_file()` are now a **single method** called `.send_file()`. It's also important to note that the **order** of the parameters has been **swapped**: first to *who* you want to send it, then the file itself.

- The same applies to `.download_msg_media()`, which has been renamed to `.download_media()`. The method now supports a `Message` itself too, rather than only `Message.media`. The specialized `.download_photo()`, `.download_document()` and `.download_contact()` still exist, but are private.

Additions

- Updated to **layer 70**!

- Both downloading and uploading now support **stream-like objects**.

- A lot **faster initial connection** if `sympy` is installed (can be installed through `pip`).

- `libssl` will also be used if available on your system (likely on Linux based systems). This speed boost should also apply to uploading and downloading files.

- You can use a **phone number** or an **username** for methods like `.send_message()`, `.send_file()`, and all the other quick-access methods provided by the `TelegramClient`.

Bug fixes

- Crashing when migrating to a new layer and receiving old updates should not happen now.

- `InputPeerChannel` is now casted to `InputChannel` automtically too.

- `.get_new_msg_id()` should now be thread-safe. No promises.

- Logging out on macOS caused a crash, which should be gone now.

- More checks to ensure that the connection is flagged correctly as either connected or not.

Note: Downloading files from CDN's will **not work** yet (something new that comes with layer 70).

That's it, any new idea or suggestion about how to make the project even more friendly is highly appreciated.

Note: Did you know that you can pretty print any result Telegram returns (called `TLObject`'s) by using their `.stringify()` function? Great for debugging!

2.30.57 get_input_* now works with vectors (v0.11.5)

Published at 2017/07/11

Quick fix-up of a bug which hadn't been encountered until now. Auto-cast by using `get_input_*` now works.

2.30.58 get_input_* everywhere (v0.11.4)

Published at 2017/07/10

For some reason, Telegram doesn't have enough with the InputPeer. There also exist InputChannel and InputUser! You don't have to worry about those anymore, it's handled internally now.

Besides this, every Telegram object now features a new default `.__str__` look, and also a .stringify() method to pretty format them, if you ever need to inspect them.

The library now uses the DEBUG level everywhere, so no more warnings or information messages if you had logging enabled.

The `no_webpage` parameter from `.send_message` has been renamed to `link_preview` for clarity, so now it does the opposite (but has a clearer intention).

2.30.59 Quick .send_message() fix (v0.11.3)

Published at 2017/07/05

A very quick follow-up release to fix a tiny bug with `.send_message()`, no new features.

2.30.60 Callable TelegramClient (v0.11.2)

Published at 2017/07/04

Scheme layer used: 68

There is a new preferred way to **invoke requests**, which you're encouraged to use:

```
# New!
result = client(SomeRequest())

# Old.
result = client.invoke(SomeRequest())
```

Existing code will continue working, since the old `.invoke()` has not been deprecated.

When you `.create_new_connection()`, it will also handle `FileMigrateError`'s for you, so you don't need to worry about those anymore.

Bugs fixes

- Fixed some errors when installing Telethon via `pip` (for those using either source distributions or a Python version 3.5).
- `ConnectionResetError` didn't flag sockets as closed, but now it does.

On a more technical side, `msg_id`'s are now more accurate.

2.30.61 Improvements to the updates (v0.11.1)

Published at 2017/06/24

Receiving new updates shouldn't miss any anymore, also, periodic pings are back again so it should work on the long run.

On a different order of things, `.connect()` also features a timeout. Notice that the `timeout=` is **not** passed as a **parameter** anymore, and is instead specified when creating the `TelegramClient`.

Bug fixes

- Fixed some name class when a request had a `.msg_id` parameter.
- The correct amount of random bytes is now used in DH request
- Fixed `CONNECTION_APP_VERSION_EMPTY` when using temporary sessions.
- Avoid connecting if already connected.

2.30.62 Support for parallel connections (v0.11)

Published at 2017/06/16

This update brings a lot of changes, so it would be nice if you could **read the whole change log**!

Breaking changes

- Every Telegram error has now its **own class**, so it's easier to fine-tune your `except`'s.

- Markdown parsing is **not part** of Telethon itself anymore, although there are plans to support it again through a some external module.

- The `.list_sessions()` has been moved to the `Session` class instead.

- The `InteractiveTelegramClient` is **not** shipped with `pip` anymore.

Additions

- A new, more **lightweight class** has been added. The `TelegramBareClient` is now the base of the normal `TelegramClient`, and has the most basic features.

- New method to `.create_new_connection()`, which can be ran **in parallel** with the original connection. This will return the previously mentioned `TelegramBareClient` already connected.

- Any file object can now be used to download a file (for instance, a `BytesIO()` instead a file name).

- Vales like `random_id` are now **automatically inferred**, so you can save yourself from the hassle of writing `generate_random_long()` everywhere. Same applies to `.get_input_peer()`, unless you really need the extra performance provided by skipping one `if` if called manually.

- Every type now features a new `.to_dict()` method.

Bug fixes

- Received errors are acknowledged to the server, so they don't happen over and over.

- Downloading media on different data centers is now up to **x2 faster**, since there used to be an `InvalidDCError` for each file part tried to be downloaded.

- Lost messages are now properly skipped.

- New way to handle the **result of requests**. The old `ValueError` "*The previously sent request must be resent. However, no request was previously sent (possibly called from a different thread).*" *should* not happen anymore.

Internal changes

- Some fixes to the `JsonSession`.

- Fixed possibly crashes if trying to `.invoke()` a `Request` while `.reconnect()` was being called on the `UpdatesThread`.

- Some improvements on the `TcpClient`, such as not switching between blocking and non-blocking sockets.

- The code now uses ASCII characters only.

- Some enhancements to `.find_user_or_chat()` and `.get_input_peer()`.

2.30.63 JSON session file (v0.10.1)

Published at 2017/06/07

This version is primarily for people to **migrate** their `.session` files, which are *pickled*, to the new *JSON* format. Although slightly slower, and a bit more vulnerable since it's plain text, it's a lot more resistant to upgrades.

> **Warning:** You **must** upgrade to this version before any higher one if you've used Telethon v0.10. If you happen
> to upgrade to an higher version, that's okay, but you will have to manually delete the `*.session` file, and logout
> from that session from an official client.

Additions

- New `.get_me()` function to get the **current** user.
- `.is_user_authorized()` is now more reliable.
- New nice button to copy the `from telethon.tl.xxx.yyy import Yyy` on the online documentation.
- **More error codes** added to the `errors` file.

Enhancements

- Everything on the documentation is now, theoretically, **sorted alphabetically**.
- No second thread is spawned unless one or more update handlers are added.

2.30.64 Full support for different DCs and ++stable (v0.10)

Published at 2017/06/03

Working with **different data centers** finally *works*! On a different order of things, **reconnection** is now performed
automatically every time Telegram decides to kick us off their servers, so now Telethon can really run **forever and
ever**! In theory.

Enhancements

- **Documentation** improvements, such as showing the return type.
- The `msg_id too low/high` error should happen **less often**, if any.
- Sleeping on the main thread is **not done anymore**. You will have to `except FloodWaitError`'s.
- You can now specify your *own application version*, device model, system version and language code.
- Code is now more *pythonic* (such as making some members private), and other internal improvements (which
  affect the **updates thread**), such as using `logger` instead a bare `print()` too.

This brings Telethon a whole step closer to `v1.0`, though more things should preferably be changed.

2.30.65 Stability improvements (v0.9.1)

Published at 2017/05/23

Telethon used to crash a lot when logging in for the very first time. The reason for this was that the reconnection (or
dead connections) were not handled properly. Now they are, so you should be able to login directly, without needing
to delete the `*.session` file anymore. Notice that downloading from a different DC is still a WIP.

Enhancements

- Updates thread is only started after a successful login.

- Files meant to be ran by the user now use **shebangs** and proper permissions.

- In-code documentation now shows the returning type.

- **Relative import** is now used everywhere, so you can rename `telethon` to anything else.

- **Dead connections** are now **detected** instead entering an infinite loop.

- **Sockets** can now be **closed** (and re-opened) properly.

- Telegram decided to update the layer 66 without increasing the number. This has been fixed and now we're up-to-date again.

2.30.66 General improvements (v0.9)

Published at 2017/05/19

> Scheme layer used: 66

Additions

- The **documentation**, available online here, has a new search bar.

- Better **cross-thread safety** by using `threading.Event`.

- More improvements for running Telethon during a **long period of time**.

Bug fixes

- **Avoid a certain crash on login** (occurred if an unexpected object ID was received).

- Avoid crashing with certain invalid UTF-8 strings.

- Avoid crashing on certain terminals by using known ASCII characters where possible.

- The `UpdatesThread` is now a daemon, and should cause less issues.

- Temporary sessions didn't actually work (with `session=None`).

Internal changes

- `.get_dialogs(count=` was renamed to `.get_dialogs(limit=`.

2.30.67 Bot login and proxy support (v0.8)

Published at 2017/04/14

Additions

- **Bot login**, thanks to @JuanPotato for hinting me about how to do it.

- **Proxy support**, thanks to @exzhawk for implementing it.

- **Logging support**, used by passing `--telethon-log=DEBUG` (or `INFO`) as a command line argument.

Bug fixes

- Connection fixes, such as avoiding connection until `.connect()` is explicitly invoked.

- Uploading big files now works correctly.

- Fix uploading big files.

- Some fixes on the updates thread, such as correctly sleeping when required.

2.30.68 Long-run bug fix (v0.7.1)

Published at 2017/02/19

If you're one of those who runs Telethon for a long time (more than 30 minutes), this update by @strayge will be great for you. It sends periodic pings to the Telegram servers so you don't get disconnected and you can still send and receive updates!

2.30.69 Two factor authentication (v0.7)

Published at 2017/01/31

Scheme layer used: 62

If you're one of those who love security the most, these are good news. You can now use two factor authentication with Telethon too! As internal changes, the coding style has been improved, and you can easily use custom session objects, and various little bugs have been fixed.

2.30.70 Updated pip version (v0.6)

Published at 2016/11/13

Scheme layer used: 57

This release has no new major features. However, it contains some small changes that make using Telethon a little bit easier. Now those who have installed Telethon via `pip` can also take advantage of changes, such as less bugs, creating empty instances of `TLObjects`, specifying a timeout and more!

2.30.71 Ready, pip, go! (v0.5)

Published at 2016/09/18

Telethon is now available as a **'Python package <https://pypi.python.org/pypi?name=Telethon>'__**! Those are really exciting news (except, sadly, the project structure had to change *a lot* to be able to do that; but hopefully it won't need to change much more, any more!)

Not only that, but more improvements have also been made: you're now able to both **sign up** and **logout**, watch a pretty "Uploading/Downloading... x%" progress, and other minor changes which make using Telethon **easier**.

2.30.72 Made InteractiveTelegramClient cool (v0.4)

Published at 2016/09/12

Yes, really cool! I promise. Even though this is meant to be a *library*, that doesn't mean it can't have a good *interactive client* for you to try the library out. This is why now you can do many, many things with the `InteractiveTelegramClient`:

- **List dialogs** (chats) and pick any you wish.
- **Send any message** you like, text, photos or even documents.
- **List** the **latest messages** in the chat.
- **Download** any message's media (photos, documents or even contacts!).
- **Receive message updates** as you talk (i.e., someone sent you a message).

It actually is an usable-enough client for your day by day. You could even add `libnotify` and pop, you're done! A great cli-client with desktop notifications.

Also, being able to download and upload media implies that you can do the same with the library itself. Did I need to mention that? Oh, and now, with even less bugs! I hope.

2.30.73 Media revolution and improvements to update handling! (v0.3)

Published at 2016/09/11

Telegram is more than an application to send and receive messages. You can also **send and receive media**. Now, this implementation also gives you the power to upload and download media from any message that contains it! Nothing can now stop you from filling up all your disk space with all the photos! If you want to, of course.

2.30.74 Handle updates in their own thread! (v0.2)

Published at 2016/09/10

This version handles **updates in a different thread** (if you wish to do so). This means that both the low level `TcpClient` and the not-so-low-level `MtProtoSender` are now multi-thread safe, so you can use them with more than a single thread without worrying!

This also implies that you won't need to send a request to **receive an update** (is someone typing? did they send me a message? has someone gone offline?). They will all be received **instantly**.

Some other cool examples of things that you can do: when someone tells you "*Hello*", you can automatically reply with another "*Hello*" without even needing to type it by yourself :)

However, be careful with spamming!! Do **not** use the program for that!

2.30.75 First working alpha version! (v0.1)

Published at 2016/09/06

Scheme layer used: 55

There probably are some bugs left, which haven't yet been found. However, the majority of code works and the application is already usable! Not only that, but also uses the latest scheme as of now *and* handles way better the errors. This tag is being used to mark this release as stable enough.

2.31 Wall of Shame

This project has an issues section for you to file **issues** whenever you encounter any when working with the library. Said section is **not** for issues on *your* program but rather issues with Telethon itself.

If you have not made the effort to 1. read through the docs and 2. look for the method you need, you will end up on the Wall of Shame, i.e. all issues labeled "RTFM":

> **rtfm** Literally "Read The F–king Manual"; a term showing the frustration of being bothered with questions so trivial that the asker could have quickly figured out the answer on their own with minimal effort, usually by reading readily-available documents. People who say"RTFM!" might be considered rude, but the true rude ones are the annoying people who take absolutely no self-responibility and expect to have all the answers handed to them personally.
>
> *"Damn, that's the twelveth time that somebody posted this question to the messageboard today! RTFM, already!"*
>
> *by Bill M. July 27, 2004*

If you have indeed read the docs, and have tried looking for the method, and yet you didn't find what you need, **that's fine**. Telegram's API can have some obscure names at times, and for this reason, there is a "question" label with questions that are okay to ask. Just state what you've tried so that we know you've made an effort, or you'll go to the Wall of Shame.

Of course, if the issue you're going to open is not even a question but a real issue with the library (thankfully, most of the issues have been that!), you won't end up here. Don't worry.

2.31.1 Current winner

The current winner is issue 213:

Issue:

alt Winner issue

Winner issue

Answer:
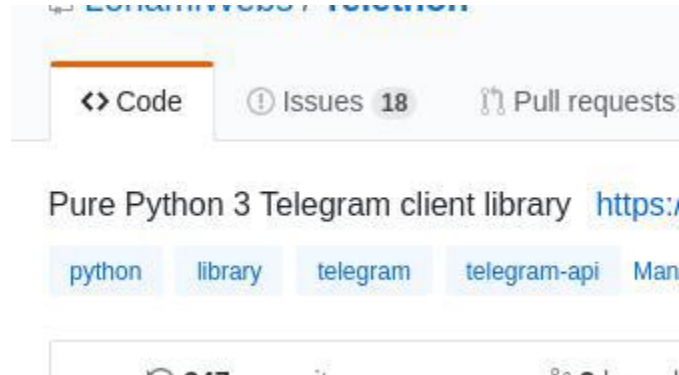
alt Winner issue answer

Winner issue answer

2.32 Compatibility and Convenience

Telethon is an `asyncio` library. Compatibility is an important concern, and while it can't always be kept and mistakes happens, the *Changelog (Version History)* is there to tell you when these important changes happen.

Contents

- *Compatibility and Convenience*
    - *Compatibility*
    - *Convenience*
    - *Speed*
    - *Learning*

2.32.1 Compatibility

Some decisions when developing will inevitable be proven wrong in the future. One of these decisions was using threads. Now that Python 3.4 is reaching EOL and using `asyncio` is usable as of Python 3.5 it makes sense for a library like Telethon to make a good use of it.

If you have old code, **just use old versions** of the library! There is nothing wrong with that other than not getting new updates or fixes, but using a fixed version with `pip install telethon==0.19.1.6` is easy enough to do.

You might want to consider using Virtual Environments in your projects.

There's no point in maintaining a synchronous version because the whole point is that people don't have time to upgrade, and there has been several changes and clean-ups. Using an older version is the right way to go.

Sometimes, other small decisions are made. These all will be reflected in the *Changelog (Version History)* which you should read when upgrading.

If you want to jump the `asyncio` boat, here are some of the things you will need to start migrating really old code:

```python
# 1. Import the client from telethon.sync
from telethon.sync import TelegramClient


# 2. Change this monster...
try:
    assert client.connect()
    if not client.is_user_authorized():
        client.send_code_request(phone_number)
        me = client.sign_in(phone_number, input('Enter code: '))

    ...   # REST OF YOUR CODE
finally:
    client.disconnect()


# ...for this:
with client:
    ...   # REST OF YOUR CODE

# 3. client.idle() no longer exists.
# Change this...
client.idle()
# ...to this:
client.run_until_disconnected()

# 4. client.add_update_handler no longer exists.
# Change this...
client.add_update_handler(handler)
# ...to this:
client.add_event_handler(handler)
```

In addition, all the update handlers must be `async def`, and you need to `await` method calls that rely on network requests, such as getting the chat or sender. If you don't use updates, you're done!

2.32.2 Convenience

Note: The entire documentation assumes you have done one of the following:

```python
from telethon import TelegramClient, sync
# or
from telethon.sync import TelegramClient
```

This makes the examples shorter and easier to think about.

For quick scripts that don't need updates, it's a lot more convenient to forget about `asyncio` and just work with sequential code. This can prove to be a powerful hybrid for running under the Python REPL too.

```python
from telethon.sync import TelegramClient
#             ^~~~~ note this part; it will manage the asyncio loop for you

with TelegramClient(...) as client:
    print(client.get_me().username)
    #     ^ notice the lack of await, or loop.run_until_complete().
    #        Since there is no loop running, this is done behind the scenes.
```

```python
    #
    message = client.send_message('me', 'Hi!')
    import time
    time.sleep(5)
    message.delete()

    # You can also have an hybrid between a synchronous
    # part and asynchronous event handlers.
    #
    from telethon import events
    @client.on(events.NewMessage(pattern='(?i)hi|hello'))
    async def handler(event):
        await event.reply('hey')

    client.run_until_disconnected()
```

Some methods, such as `with`, `start`, `disconnect` and `run_until_disconnected` work both in synchronous and asynchronous contexts by default for convenience, and to avoid the little overhead it has when using methods like sending a message, getting messages, etc. This keeps the best of both worlds as a sane default.

Note: As a rule of thumb, if you're inside an `async def` and you need the client, you need to `await` calls to the API. If you call other functions that also need API calls, make them `async def` and `await` them too. Otherwise, there is no need to do so with this mode.

2.32.3 Speed

When you're ready to micro-optimize your application, or if you simply don't need to call any non-basic methods from a synchronous context, just get rid of `telethon.sync` and work inside an `async def`:

```python
import asyncio
from telethon import TelegramClient, events

async def main():
    async with TelegramClient(...) as client:
        print((await client.get_me()).username)
        #       ^_____^ notice these parenthesis
        #       You want to ``await`` the call, not the username.
        #
        message = await client.send_message('me', 'Hi!')
        await asyncio.sleep(5)
        await message.delete()

        @client.on(events.NewMessage(pattern='(?i)hi|hello'))
        async def handler(event):
            await event.reply('hey')

        await client.run_until_disconnected()

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

The `telethon.sync` magic module simply wraps every method behind:

```
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

So that you don't have to write it yourself every time. That's the overhead you pay if you import it, and what you save if you don't.

2.32.4 Learning

You know the library uses `asyncio` everywhere, and you want to learn how to do things right. Even though `asyncio` is its own topic, the documentation wants you to learn how to use Telethon correctly, and for that, you need to use `asyncio` correctly too. For this reason, there is a section called *Mastering asyncio* that will introduce you to the `asyncio` world, with links to more resources for learning how to use it. Feel free to check that section out once you have read the rest.

2.33 TelegramClient

The *TelegramClient* aggregates several mixin classes to provide all the common functionality in a nice, Pythonic interface. Each mixin has its own methods, which you all can use.

In short, to create a client you must run:

```python
import asyncio
from telethon import TelegramClient

async def main():
    client = await TelegramClient(name, api_id, api_hash).start()
    # Now you can use all client methods listed below, like for example...
    await client.send_message('me', 'Hello to myself!')

asyncio.get_event_loop().run_until_complete(main())
```

You **don't** need to import these `AuthMethods`, `MessageMethods`, etc. Together they are the *TelegramClient* and you can access all of their methods.

See *Client Reference* for a short summary.

class telethon.client.telegramclient.**TelegramClient**(*session: typing.Union[str, Session], api_id: int, api_hash: str, *, connection: typing.Type[Connection] = <class 'telethon.network.connection.tcpfull.ConnectionTcpFull'>, use_ipv6: bool = False, proxy: Union[tuple, dict] = None, timeout: int = 10, request_retries: int = 5, connection_retries: int = 5, retry_delay: int = 1, auto_reconnect: bool = True, sequential_updates: bool = False, flood_sleep_threshold: int = 60, device_model: str = None, system_version: str = None, app_version: str = None, lang_code: str = 'en', system_lang_code: str = 'en', loop: asyncio.events.AbstractEventLoop = None, base_logger: Union[str, logging.Logger] = None*)

    Bases: [*telethon.client.account.AccountMethods*](), [*telethon.client.auth.AuthMethods*](), [*telethon.client.downloads.DownloadMethods*](), [*telethon.client.dialogs.DialogMethods*](), [*telethon.client.chats.ChatMethods*](), [*telethon.client.bots.BotMethods*](), [*telethon.client.messages.MessageMethods*](), [*telethon.client.uploads.UploadMethods*](), [*telethon.client.buttons.ButtonMethods*](), [*telethon.client.updates.UpdateMethods*](), [*telethon.client.messageparse.MessageParseMethods*](), [*telethon.client.users.UserMethods*](), [*telethon.client.telegrambaseclient.TelegramBaseClient*]()

class telethon.client.telegrambaseclient.**TelegramBaseClient**(*session:* *typ-ing.Union[str,* *Session],* *api_id:* *int, api_hash: str,* **, connection: typ-ing.Type[Connection]* *=* *<class* *'telethon.network.connection.tcpfull.Connecti* *use_ipv6:* *bool* *=* *False,* *proxy:* *Union[tuple, dict] =* *None, timeout: int =* *10, request_retries:* *int = 5, connec-tion_retries: int =* *5, retry_delay: int =* *1, auto_reconnect:* *bool = True, se-quential_updates:* *bool* *=* *False,* *flood_sleep_threshold:* *int = 60, de-vice_model:* *str* *=* *None,* *sys-tem_version:* *str* *=* *None,* *app_version:* *str* *= None, lang_code:* *str = 'en', sys-tem_lang_code: str* *= 'en', loop: asyn-cio.events.AbstractEventLoop* *=* *None,* *base_logger:* *Union[str,* *log-ging.Logger]* *=* *None*)

Bases: abc.ABC

This is the abstract base class for the client. It defines some basic stuff like connecting, switching data center, etc, and leaves the *__call__* unimplemented.

Arguments

> **session** (str | *telethon.sessions.abstract.Session*, None): The file name of the session file to be used if a string is given (it may be a full path), or the Session instance to be used otherwise. If it's None, the session will not be saved, and you should call *log_out()* when you're done.
>
> Note that if you pass a string it will be a file in the current working directory, although you can also pass absolute paths.
>
> The session file contains enough information for you to login without re-sending the code, so if you have to enter the code more than once, maybe you're changing the working directory, renaming or removing the file, or using random names.
>
> **api_id** (int | str): The API ID you obtained from https://my.telegram.org.
>
> **api_hash** (str): The API ID you obtained from https://my.telegram.org.

connection (`telethon.network.connection.common.Connection`, optional): The connection instance to be used when creating a new connection to the servers. It **must** be a type.

> Defaults to *`telethon.network.connection.tcpfull.ConnectionTcpFull`*.

use_ipv6 (`bool`, optional): Whether to connect to the servers through IPv6 or not. By default this is `False` as IPv6 support is not too widespread yet.

proxy (`tuple` | `list` | `dict`, optional): An iterable consisting of the proxy info. If `connection` is one of MTProxy, then it should contain MTProxy credentials: `('hostname', port, 'secret')`. Otherwise, it's meant to store function parameters for PySocks, like `(type, 'hostname', port)`. See https://github.com/Anorov/PySocks#usage-1 for more.

timeout (`int` | `float`, optional): The timeout in seconds to be used when connecting. This is **not** the timeout to be used when `await`'ing for invoked requests, and you should use `asyncio.wait` or `asyncio.wait_for` for that.

request_retries (`int` | `None`, optional): How many times a request should be retried. Request are retried when Telegram is having internal issues (due to either `errors.ServerError` or `errors.RpcCallFailError`), when there is a `errors.FloodWaitError` less than `flood_sleep_threshold`, or when there's a migrate error.

> May take a negative or `None` value for infinite retries, but this is not recommended, since some requests can always trigger a call fail (such as searching for messages).

connection_retries (`int` | `None`, optional): How many times the reconnection should retry, either on the initial connection or when Telegram disconnects us. May be set to a negative or `None` value for infinite retries, but this is not recommended, since the program can get stuck in an infinite loop.

retry_delay (`int` | `float`, optional): The delay in seconds to sleep between automatic reconnections.

auto_reconnect (`bool`, optional): Whether reconnection should be retried `connection_retries` times automatically if Telegram disconnects us or not.

sequential_updates (`bool`, optional): By default every incoming update will create a new task, so you can handle several updates in parallel. Some scripts need the order in which updates are processed to be sequential, and this setting allows them to do so.

> If set to `True`, incoming updates will be put in a queue and processed sequentially. This means your event handlers should *not* perform long-running operations since new updates are put inside of an unbounded queue.

flood_sleep_threshold (`int` | `float`, optional): The threshold below which the library should automatically sleep on flood wait errors (inclusive). For instance, if a `FloodWaitError` for 17s occurs and `flood_sleep_threshold` is 20s, the library will `sleep` automatically. If the error was for 21s, it would `raise FloodWaitError` instead. Values larger than a day (like `float('inf')`) will be changed to a day.

device_model (`str`, optional): "Device model" to be sent when creating the initial connection. Defaults to `platform.node()`.

system_version (`str`, optional): "System version" to be sent when creating the initial connection. Defaults to `platform.system()`.

app_version (`str`, optional): "App version" to be sent when creating the initial connection. Defaults to `telethon.version.__version__`.

lang_code (`str`, optional): "Language code" to be sent when creating the initial connection. Defaults to `'en'`.

system_lang_code (`str`, optional): "System lang code" to be sent when creating the initial connection. Defaults to `lang_code`.

> **loop** (`asyncio.AbstractEventLoop`, optional): Asyncio event loop to use. Defaults to `asyncio.get_event_loop()`
>
> **base_logger** (`str` | `logging.Logger`, optional): Base logger name or instance to use. If a `str` is given, it'll be passed to `logging.getLogger()`. If a `logging.Logger` is given, it'll be used directly. If something else or nothing is given, the default logger will be used.

__call__(*request*, *ordered=False*)
> Invokes (sends) one or more MTProtoRequests and returns (receives) their result.
>
> **Args:**
>
> > **request** (`TLObject` | `list`): The request or requests to be invoked.
> >
> > **ordered** (`bool`, optional): Whether the requests (if more than one was given) should be executed sequentially on the server. They run in arbitrary order by default.
>
> **Returns:** The result of the request (often a `TLObject`) or a list of results if more than one request was given.

__dict__ = mappingproxy({'__module__': 'telethon.client.telegrambaseclient', '__doc__**

__version__ = '1.9.0'

__weakref__
> list of weak references to the object (if defined)

connect() → None
> Connects to Telegram.

Note: Connect means connect and nothing else, and only one low-level request is made to notify Telegram about which layer we will be using.

Before Telegram sends you updates, you need to make a high-level request, like *client.get_me()*, as described in https://core.telegram.org/api/updates.

> **Example**
>
> ```
> try:
>     client.connect()
> except OSError:
>     print('Failed to connect')
> ```

disconnect()
> Disconnects from Telegram.
>
> If the event loop is already running, this method returns a coroutine that you should await on your own code; otherwise the loop is ran until said coroutine completes.
>
> **Example**
>
> ```
> # You don't need to use this if you used "with client"
> client.disconnect()
> ```

disconnected
> Property with a `Future` that resolves upon disconnection.
>
> **Example**

```
# Wait for a disconnection to occur
try:
    await client.disconnected
except OSError:
    print('Error on disconnect')
```

is_connected() → bool

> Returns [True](#) if the user has connected.
>
> This method is **not** asynchronous (don't use await on it).
>
> **Example**
>
> ```
> while client.is_connected():
>     await asyncio.sleep(1)
> ```

loop

> Property with the asyncio event loop used by this client.
>
> **Example**
>
> ```
> # Download media in the background
> task = client.loop_create_task(message.download_media())
>
> # Do some work
> ...
>
> # Join the task (wait for it to complete)
> await task
> ```

class telethon.client.account.**AccountMethods**

> Bases: [object](#)
>
> **__dict__** = mappingproxy({'__module__': 'telethon.client.account', 'takeout': <functi
>
> **__weakref__**
>
> > list of weak references to the object (if defined)
>
> **end_takeout**(*success: bool*) → bool
>
> > Finishes the current takeout session.
> >
> > **Arguments**
> >
> > > success ([bool](#)): Whether the takeout completed successfully or not.
> >
> > **Returns** [True](#) if the operation was successful, [False](#) otherwise.
> >
> > **Example**
> >
> > ```
> > client.end_takeout(success=False)
> > ```
>
> **takeout**(*finalize: bool = True*, ***, *contacts: bool = None*, *users: bool = None*, *chats: bool = None*, *megagroups: bool = None*, *channels: bool = None*, *files: bool = None*, *max_file_size: bool = None*) → TelegramClient
>
> > Returns a *[TelegramClient](#)* which calls methods behind a takeout session.
> >
> > It does so by creating a proxy object over the current client through which making requests will use [InvokeWithTakeoutRequest](#) to wrap them. In other words, returns the current client modified so that requests are done as a takeout:
> >
> > Some of the calls made through the takeout session will have lower flood limits. This is useful if you want to export the data from conversations or mass-download media, since the rate limits will be lower. Only

some requests will be affected, and you will need to adjust the `wait_time` of methods like `client.iter_messages`.

By default, all parameters are `None`, and you need to enable those you plan to use by setting them to either `True` or `False`.

You should `except errors.TakeoutInitDelayError as e`, since this exception will raise depending on the condition of the session. You can then access `e.seconds` to know how long you should wait for before calling the method again.

There's also a `success` property available in the takeout proxy object, so from the `with` body you can set the boolean result that will be sent back to Telegram. But if it's left `None` as by default, then the action is based on the `finalize` parameter. If it's `True` then the takeout will be finished, and if no exception occurred during it, then `True` will be considered as a result. Otherwise, the takeout will not be finished and its ID will be preserved for future usage as `client.session.takeout_id`.

Arguments

> **finalize (`bool`):** Whether the takeout session should be finalized upon exit or not.
>
> **contacts (`bool`):** Set to `True` if you plan on downloading contacts.
>
> **users (`bool`):** Set to `True` if you plan on downloading information from users and their private conversations with you.
>
> **chats (`bool`):** Set to `True` if you plan on downloading information from small group chats, such as messages and media.
>
> **megagroups (`bool`):** Set to `True` if you plan on downloading information from megagroups (channels), such as messages and media.
>
> **channels (`bool`):** Set to `True` if you plan on downloading information from broadcast channels, such as messages and media.
>
> **files (`bool`):** Set to `True` if you plan on downloading media and you don't only wish to export messages.
>
> **max_file_size (`int`):** The maximum file size, in bytes, that you plan to download for each message with media.

Example

```python
from telethon import errors

try:
    with client.takeout() as takeout:
        client.get_messages('me')  # normal call
        takeout.get_messages('me')  # wrapped through takeout (less
    limits)

        for message in takeout.iter_messages(chat, wait_time=0):
            ...  # Do something with the message

except errors.TakeoutInitDelayError as e:
    print('Must wait', e.seconds, 'before takeout')
```

class `telethon.client.auth.`**`AuthMethods`**

> Bases: `object`

`__aenter__``()`

`__aexit__``(*args)`

__dict__ = mappingproxy({'**__module__**': '**telethon.client.auth**', '**start**': **<function Au**

__enter__()
> Helps to cut boilerplate on async context managers that offer synchronous variants.

__exit__(**args*)

__weakref__
> list of weak references to the object (if defined)

edit_2fa(*current_password: str = None, new_password: str = None, *, hint: str = '', email: str = None, email_code_callback: Callable[[int], str] = None*) → bool
Changes the 2FA settings of the logged in user.

> Review carefully the parameter explanations before using this method.

> Note that this method may be *incredibly* slow depending on the prime numbers that must be used during the process to make sure that everything is safe.

> Has no effect if both current and new password are omitted.

> **Arguments**

>> **current_password (`str`, optional):** The current password, to authorize changing to `new_password`. Must be set if changing existing 2FA settings. Must **not** be set if 2FA is currently disabled. Passing this by itself will remove 2FA (if correct).

>> **new_password (`str`, optional):** The password to set as 2FA. If 2FA was already enabled, `current_password` **must** be set. Leaving this blank or `None` will remove the password.

>> **hint (`str`, optional):** Hint to be displayed by Telegram when it asks for 2FA. Leaving unspecified is highly discouraged. Has no effect if `new_password` is not set.

>> **email (`str`, optional):** Recovery and verification email. If present, you must also set `email_code_callback`, else it raises `ValueError`.

>> **email_code_callback (`callable`, optional):** If an email is provided, a callback that returns the code sent to it must also be set. This callback may be asynchronous. It should return a string with the code. The length of the code will be passed to the callback as an input parameter.

>> If the callback returns an invalid code, it will raise `CodeInvalidError`.

> **Returns** `True` if successful, `False` otherwise.

> **Example**

> ```python
> # Setting a password for your account which didn't have
> client.edit_2fa(new_password='I_<3_Telethon')
>
> # Removing the password
> client.edit_2fa(current_password='I_<3_Telethon')
> ```

log_out() → bool
> Logs out Telegram and deletes the current `*.session` file.

> **Returns** `True` if the operation was successful.

> **Example**

> ```python
> # Note: you will need to login again!
> client.log_out()
> ```

send_code_request(*phone: str, *, force_sms: bool = False*) → types.auth.SentCode
Sends the Telegram code needed to login to the given phone number.

Arguments

> **phone (`str` | `int`):** The phone to which the code will be sent.
>
> **force_sms (`bool`, optional):** Whether to force sending as SMS.

Returns An instance of SentCode.

Example

```
phone = '+34 123 123 123'
sent = client.send_code_request(phone)
print(sent)

if sent.phone_registered:
    print('This phone has an existing account registered')
else:
    print('This phone does not have an account registered')
```

sign_in (*phone: str = None*, *code: Union[str, int] = None*, ***, *password: str = None*, *bot_token: str = None*, *phone_code_hash: str = None*) → types.User
Logs in to Telegram to an existing user or bot account.

You should only use this if you are not authorized yet.

This method will send the code if it's not provided.

Note: In most cases, you should simply use *start()* and not this method.

Arguments

> **phone (`str` | `int`):** The phone to send the code to if no code was provided, or to override the phone that was previously used with these requests.
>
> **code (`str` | `int`):** The code that Telegram sent. Note that if you have sent this code through the application itself it will immediately expire. If you want to send the code, obfuscate it somehow. If you're not doing any of this you can ignore this note.
>
> **password (`str`):** 2FA password, should be used if a previous call raised `SessionPasswordNeededError`.
>
> **bot_token (`str`):** Used to sign in as a bot. Not all requests will be available. This should be the hash the @BotFather gave you.
>
> **phone_code_hash (`str`, optional):** The hash returned by *send_code_request*. This can be left as None to use the last hash known for the phone to be used.

Returns The signed in user, or the information about *send_code_request()*.

Example

```
phone = '+34 123 123 123'
client.sign_in(phone)  # send code

code = input('enter code: ')
client.sign_in(phone, code)
```

sign_up (*code: Union[str, int]*, *first_name: str*, *last_name: str = ''*, ***, *phone: str = None*, *phone_code_hash: str = None*) → types.User
Signs up to Telegram as a new user account.

Use this if you don't have an account yet.

You must call *send_code_request* first.

By using this method you're agreeing to Telegram's Terms of Service. This is required and your account will be banned otherwise. See https://telegram.org/tos and https://core.telegram.org/api/terms.

Arguments

> **code (`str` | `int`):** The code sent by Telegram
>
> **first_name (`str`):** The first name to be used by the new account.
>
> **last_name (`str`, optional)** Optional last name.
>
> **phone (`str` | `int`, optional):** The phone to sign up. This will be the last phone used by default (you normally don't need to set this).
>
> **phone_code_hash (`str`, optional):** The hash returned by *send_code_request*. This can be left as `None` to use the last hash known for the phone to be used.

Returns The new created User.

Example

```
phone = '+34 123 123 123'
client.send_code_request(phone)

code = input('enter code: ')
client.sign_up(code, first_name='Anna', last_name='Banana')
```

start (*phone: Callable[[], str] = <function AuthMethods.<lambda>>, password: Callable[[], str] = <function AuthMethods.<lambda>>, *, bot_token: str = None, force_sms: bool = False, code_callback: Callable[[], Union[str, int]] = None, first_name: str = 'New User', last_name: str = '', max_attempts: int = 3*) → TelegramClient
Starts the client (connects and logs in if necessary).

By default, this method will be interactive (asking for user input if needed), and will handle 2FA if enabled too.

If the phone doesn't belong to an existing account (and will hence *sign_up* for a new one), **you are agreeing to Telegram's Terms of Service. This is required and your account will be banned otherwise.** See https://telegram.org/tos and https://core.telegram.org/api/terms.

If the event loop is already running, this method returns a coroutine that you should await on your own code; otherwise the loop is ran until said coroutine completes.

Arguments

> **phone (`str` | `int` | `callable`):** The phone (or callable without arguments to get it) to which the code will be sent. If a bot-token-like string is given, it will be used as such instead. The argument may be a coroutine.
>
> **password (`str`, `callable`, optional):** The password for 2 Factor Authentication (2FA). This is only required if it is enabled in your account. The argument may be a coroutine.
>
> **bot_token (`str`):** Bot Token obtained by @BotFather to log in as a bot. Cannot be specified with `phone` (only one of either allowed).
>
> **force_sms (`bool`, optional):** Whether to force sending the code request as SMS. This only makes sense when signing in with a `phone`.
>
> **code_callback (`callable`, optional):** A callable that will be used to retrieve the Telegram login code. Defaults to `input()`. The argument may be a coroutine.

> **first_name** (`str`, optional): The first name to be used if signing up. This has no effect if the account already exists and you sign in.
>
> **last_name** (`str`, optional): Similar to the first name, but for the last. Optional.
>
> **max_attempts** (`int`, optional): How many times the code/password callback should be retried or switching between signing in and signing up.

> **Returns** This `TelegramClient`, so initialization can be chained with `.start()`.

> **Example**

```
client = TelegramClient('anon', api_id, api_hash)

# Starting as a bot account
client.start(bot_token=bot_token)

# Starting as an user account
client.start(phone)
# Please enter the code you received: 12345
# Please enter your password: *******
# (You are now logged in)

# Starting using a context manager (this calls start()):
with client:
    pass
```

class telethon.client.bots.**BotMethods**

> Bases: `object`

> **__dict__** = mappingproxy({'__module__': 'telethon.client.bots', 'inline_query': <func

> **__weakref__**
> list of weak references to the object (if defined)

> **inline_query**(*bot: hints.EntityLike*, *query: str*, ***, *offset: str = None*, *geo_point: types.GeoPoint = None*) → telethon.tl.custom.inlineresults.InlineResults
> Makes an inline query to the specified bot (e.g. @vote New Poll).

> > **Arguments**

> > > **bot** (`entity`): The bot entity to which the inline query should be made.

> > > **query** (`str`): The query that should be made to the bot.

> > > **offset** (`str`, optional): The string offset to use for the bot.

> > > **geo_point** (GeoPoint, optional) The geo point location information to send to the bot for localised results. Available under some bots.

> > **Returns** A list of *custom.InlineResult*.

> > **Example**

```
# Make an inline query to @like
results = client.inline_query('like', 'Do you like Telethon?')

# Send the first result to some chat
message = results[0].click('TelethonOffTopic')
```

class telethon.client.buttons.**ButtonMethods**

> Bases: `object`

> **__dict__** = mappingproxy({'__module__': 'telethon.client.buttons', 'build_reply_markup

__weakref__
    list of weak references to the object (if defined)

static build_reply_markup(*buttons:*   *Union[telethon.tl.types.ReplyKeyboardHide, telethon.tl.types.ReplyKeyboardForceReply, telethon.tl.types.ReplyKeyboardMarkup, telethon.tl.types.ReplyInlineMarkup, telethon.tl.types.KeyboardButton, telethon.tl.types.KeyboardButtonUrl, telethon.tl.types.KeyboardButtonCallback, telethon.tl.types.KeyboardButtonRequestPhone, telethon.tl.types.KeyboardButtonRequestGeoLocation, telethon.tl.types.KeyboardButtonSwitchInline, telethon.tl.types.KeyboardButtonGame, telethon.tl.types.KeyboardButtonBuy, telethon.tl.types.KeyboardButtonUrlAuth, telethon.tl.types.InputKeyboardButtonUrlAuth, telethon.tl.custom.button.Button,*   *Sequence[Union[telethon.tl.types.KeyboardButton, telethon.tl.types.KeyboardButtonUrl, telethon.tl.types.KeyboardButtonCallback, telethon.tl.types.KeyboardButtonRequestPhone, telethon.tl.types.KeyboardButtonRequestGeoLocation, telethon.tl.types.KeyboardButtonSwitchInline, telethon.tl.types.KeyboardButtonGame, telethon.tl.types.KeyboardButtonBuy, telethon.tl.types.KeyboardButtonUrlAuth, telethon.tl.types.InputKeyboardButtonUrlAuth, telethon.tl.custom.button.Button]],*   *Sequence[Sequence[Union[telethon.tl.types.KeyboardButton, telethon.tl.types.KeyboardButtonUrl, telethon.tl.types.KeyboardButtonCallback, telethon.tl.types.KeyboardButtonRequestPhone, telethon.tl.types.KeyboardButtonRequestGeoLocation, telethon.tl.types.KeyboardButtonSwitchInline, telethon.tl.types.KeyboardButtonGame, telethon.tl.types.KeyboardButtonBuy, telethon.tl.types.KeyboardButtonUrlAuth, telethon.tl.types.InputKeyboardButtonUrlAuth, telethon.tl.custom.button.Button]]], None], inline_only: bool = False*) → Union[telethon.tl.types.ReplyKeyboardHide, telethon.tl.types.ReplyKeyboardForceReply, telethon.tl.types.ReplyKeyboardMarkup, telethon.tl.types.ReplyInlineMarkup, None]

Builds a ReplyInlineMarkup or ReplyKeyboardMarkup for the given buttons.

Does nothing if either no buttons are provided or the provided argument is already a reply markup.

You should consider using this method if you are going to reuse the markup very often. Otherwise, it is not necessary.

This method is **not** asynchronous (don't use `await` on it).

Arguments

    buttons (`hints.MarkupLike`): The button, list of buttons, array of buttons or markup to convert into a markup.

    inline_only (`bool`, optional): Whether the buttons **must** be inline buttons only or not.

Example

```python
from telethon import Button

markup = client.build_reply_markup(Button.inline('hi'))
client.send_message('click me', buttons=markup)
```

class telethon.client.chats.**ChatMethods**

    Bases: object

    __dict__ = mappingproxy({'__module__': 'telethon.client.chats', 'iter_participants':

    __weakref__

        list of weak references to the object (if defined)

    action(*entity: hints.EntityLike, action: typing.Union[str, types.TypeSendMessageAction], *, delay:*
        float = 4, auto_cancel: bool = True) → typing.Union[_ChatAction, typing.Coroutine]
        Returns a context-manager object to represent a "chat action".

    Chat actions indicate things like "user is typing", "user is uploading a photo", etc.

    If the action is 'cancel', you should just await the result, since it makes no sense to use a context-manager for it.

    See the example below for intended usage.

    Arguments

        entity (entity): The entity where the action should be showed in.

        action (str | SendMessageAction): The action to show. You can either pass a instance of SendMessageAction or better, a string used while:

            • 'typing': typing a text message.

            • 'contact': choosing a contact.

            • 'game': playing a game.

            • 'location': choosing a geo location.

            • 'record-audio': recording a voice note. You may use 'record-voice' as alias.

            • 'record-round': recording a round video.

            • 'record-video': recording a normal video.

            • 'audio': sending an audio file (voice note or song). You may use 'voice' and 'song' as aliases.

            • 'round': uploading a round video.

            • 'video': uploading a video file.

            • 'photo': uploading a photo.

            • 'document': uploading a document file. You may use 'file' as alias.

            • 'cancel': cancel any pending action in this chat.

        Invalid strings will raise a ValueError.

        delay (int | float): The delay, in seconds, to wait between sending actions. For example, if the delay is 5 and it takes 7 seconds to do something, three requests will be made at 0s, 5s, and 7s to cancel the action.

> **auto_cancel (`bool`):** Whether the action should be cancelled once the context manager exists or not.
> The default is `True`, since you don't want progress to be shown when it has already completed.

Returns Either a context-manager object or a coroutine.

Example

```python
# Type for 2 seconds, then send a message
async with client.action(chat, 'typing'):
    await asyncio.sleep(2)
    await client.send_message(chat, 'Hello world! I type slow ^^')

# Cancel any previous action
await client.action(chat, 'cancel')

# Upload a document, showing its progress (most clients ignore this)
async with client.action(chat, 'document') as action:
    client.send_file(chat, zip_file, progress_callback=action.progress)
```

edit_admin(*entity: hints.EntityLike, user: hints.EntityLike, *, change_info: bool = None, post_messages: bool = None, edit_messages: bool = None, delete_messages: bool = None, ban_users: bool = None, invite_users: bool = None, pin_messages: bool = None, add_admins: bool = None, is_admin: bool = None*) → telethon.tl.types.Updates

Edits admin permissions for someone in a chat.

Raises an error if wrong a combination of rights are given (e.g. you don't have enough permissions to grant one).

Unless otherwise stated, permissions will work in channels and megagroups.

Arguments

> **entity (`entity`):** The channel, megagroup or chat where the promotion should happen.
>
> **user (`entity`):** The user to be promoted.
>
> **change_info (`bool`, optional):** Whether the user will be able to change info.
>
> **post_messages (`bool`, optional):** Whether the user will be able to post in the channel. This will only work in broadcast channels.
>
> **edit_messages (`bool`, optional):** Whether the user will be able to edit messages in the channel. This will only work in broadcast channels.
>
> **delete_messages (`bool`, optional):** Whether the user will be able to delete messages.
>
> **ban_users (`bool`, optional):** Whether the user will be able to ban users.
>
> **invite_users (`bool`, optional):** Whether the user will be able to invite users. Needs some testing.
>
> **pin_messages (`bool`, optional):** Whether the user will be able to pin messages.
>
> **add_admins (`bool`, optional):** Whether the user will be able to add admins.
>
> **is_admin (`bool`, optional):** Whether the user will be an admin in the chat. This will only work in small group chats. Whether the user will be an admin in the chat. This is the only permission available in small group chats, and when used in megagroups, all non-explicitly set permissions will have this value.
>
> Essentially, only passing `is_admin=True` will grant all permissions, but you can still disable those you need.

Returns The resulting Updates object.

Example

```
# Allowing `user` to pin messages in `chat`
client.edit_admin(chat, user, pin_messages=True)

# Granting all permissions except for `add_admins`
client.edit_admin(chat, user, is_admin=True, add_admins=False)
```

edit_permissions(*entity: hints.EntityLike, user: typing.Optional[hints.EntityLike] = None, until_date: hints.DateLike = None, *, view_messages: bool = True, send_messages: bool = True, send_media: bool = True, send_stickers: bool = True, send_gifs: bool = True, send_games: bool = True, send_inline: bool = True, send_polls: bool = True, change_info: bool = False, invite_users: bool = False, pin_messages: bool = False*) → telethon.tl.types.Updates

Edits user restrictions in a chat.

Raises an error if wrong a combination of rights are given (e.g. you don't have enough permissions to revoke one).

Each of the boolean arguments answer the question "can the given user do this?", which by default is `True` for all permissions except those which state otherwise.

Arguments

> **entity (`entity`):** The channel or megagroup where the restriction should happen.
>
> **user (`entity`, optional):** If specified, the permission will be changed for the specific user. If left as `None`, the default chat permissions will be updated.
>
> **until_date (`DateLike`, optional):** When the user will be unbanned.
>
> > If the due date or duration is longer than 366 days or shorter than 30 seconds, the ban will be forever. Defaults to `0` (ban forever).
>
> **view_messages (`bool`, optional):** Whether the user is able to view messages or not. Forbidding someone from viewing messages equals to banning them. This will only work if `user` is set.
>
> **send_messages (`bool`, optional):** Whether the user is able to send messages or not.
>
> **send_media (`bool`, optional):** Whether the user is able to send media or not.
>
> **send_stickers (`bool`, optional):** Whether the user is able to send stickers or not.
>
> **send_gifs (`bool`, optional):** Whether the user is able to send animated gifs or not.
>
> **send_games (`bool`, optional):** Whether the user is able to send games or not.
>
> **send_inline (`bool`, optional):** Whether the user is able to use inline bots or not.
>
> **send_polls (`bool`, optional):** Whether the user is able to send polls or not.
>
> **change_info (`bool`, optional):** Whether the user is able to change info or not. Unlike the rest of the arguments, this is `False` by default.
>
> **invite_users (`bool`, optional):** Whether the user is able to invite other users or not. Unlike the rest of the arguments, this is `False` by default.
>
> **pin_messages (`bool`, optional):** Whether the user is able to pin messages or not. Unlike the rest of the arguments, this is `False` by default.

Returns The resulting Updates object.

Example

```python
from datetime import timedelta

# Kicking `user` from `chat` for 1 minute
client.edit_permission(chat, user, timedelta(minutes=1),
                       view_messages=False)

# Banning `user` from `chat` forever
client.edit_permission(chat, user, view_messages=False)
```

get_admin_log(**args*, ***kwargs*) → hints.TotalList

    Same as *iter_admin_log()*, but returns a `list` instead.

    Example

```python
# Get a list of deleted message events which said "heck"
events = client.get_admin_log(channel, search='heck', delete=True)

# Print the old message before it was deleted
print(events[0].old)
```

get_participants(**args*, ***kwargs*) → hints.TotalList

    Same as *iter_participants()*, but returns a *TotalList* instead.

    Example

```python
users = client.get_participants(chat)
print(users[0].first_name)

for user in users:
    if user.username is not None:
        print(user.username)
```

get_profile_photos(**args*, ***kwargs*) → hints.TotalList

    Same as *iter_profile_photos()*, but returns a *TotalList* instead.

    Example

```python
# Get the photos of a channel
photos = client.get_profile_photos(channel)

# Download the oldest photo
client.download_media(photos[-1])
```

iter_admin_log(*entity: hints.EntityLike*, *limit: float = None*, ***, *max_id: int = 0*, *min_id: int = 0*, *search: str = None*, *admins: hints.EntitiesLike = None*, *join: bool = None*, *leave: bool = None*, *invite: bool = None*, *restrict: bool = None*, *unrestrict: bool = None*, *ban: bool = None*, *unban: bool = None*, *promote: bool = None*, *demote: bool = None*, *info: bool = None*, *settings: bool = None*, *pinned: bool = None*, *edit: bool = None*, *delete: bool = None*) → telethon.client.chats._AdminLogIter

    Iterator over the admin log for the specified channel.

    The default order is from the most recent event to to the oldest.

    Note that you must be an administrator of it to use this method.

    If none of the filters are present (i.e. they all are `None`), *all* event types will be returned. If at least one of them is `True`, only those that are true will be returned.

    Arguments

> **entity (`entity`):** The channel entity from which to get its admin log.
>
> **limit (`int` | `None`, optional):** Number of events to be retrieved.
>
>> The limit may also be `None`, which would eventually return the whole history.
>
> **max_id (`int`):** All the events with a higher (newer) ID or equal to this will be excluded.
>
> **min_id (`int`):** All the events with a lower (older) ID or equal to this will be excluded.
>
> **search (`str`):** The string to be used as a search query.
>
> **admins (`entity` | `list`):** If present, the events will be filtered by these admins (or single admin) and only those caused by them will be returned.
>
> **join (`bool`):** If `True`, events for when a user joined will be returned.
>
> **leave (`bool`):** If `True`, events for when a user leaves will be returned.
>
> **invite (`bool`):** If `True`, events for when a user joins through an invite link will be returned.
>
> **restrict (`bool`):** If `True`, events with partial restrictions will be returned. This is what the API calls "ban".
>
> **unrestrict (`bool`):** If `True`, events removing restrictions will be returned. This is what the API calls "unban".
>
> **ban (`bool`):** If `True`, events applying or removing all restrictions will be returned. This is what the API calls "kick" (restricting all permissions removed is a ban, which kicks the user).
>
> **unban (`bool`):** If `True`, events removing all restrictions will be returned. This is what the API calls "unkick".
>
> **promote (`bool`):** If `True`, events with admin promotions will be returned.
>
> **demote (`bool`):** If `True`, events with admin demotions will be returned.
>
> **info (`bool`):** If `True`, events changing the group info will be returned.
>
> **settings (`bool`):** If `True`, events changing the group settings will be returned.
>
> **pinned (`bool`):** If `True`, events of new pinned messages will be returned.
>
> **edit (`bool`):** If `True`, events of message edits will be returned.
>
> **delete (`bool`):** If `True`, events of message deletions will be returned.

Yields Instances of *AdminLogEvent*.

Example

```
for event in client.iter_admin_log(channel):
    if event.changed_title:
        print('The title changed from', event.old, 'to', event.new)
```

iter_participants(*entity: hints.EntityLike*, *limit: float = None*, *, *search: str = ''*, *filter: types.TypeChannelParticipantsFilter = None*, *aggressive: bool = False*) → telethon.client.chats._ParticipantsIter

Iterator over the participants belonging to the specified chat.

The order is unspecified.

Arguments

> **entity (`entity`):** The entity from which to retrieve the participants list.
>
> **limit (`int`):** Limits amount of participants fetched.

search (`str`, optional): Look for participants with this string in name/username.

If `aggressive is True`, the symbols from this string will be used.

filter (ChannelParticipantsFilter, optional): The filter to be used, if you want e.g. only admins Note that you might not have permissions for some filter. This has no effect for normal chats or users.

Note: The filter ChannelParticipantsBanned will return *restricted* users. If you want *banned* users you should use ChannelParticipantsKicked instead.

---

aggressive (`bool`, optional): Aggressively looks for all participants in the chat.

This is useful for channels since 20 July 2018, Telegram added a server-side limit where only the first 200 members can be retrieved. With this flag set, more than 200 will be often be retrieved.

This has no effect if a `filter` is given.

Yields The User objects returned by GetParticipantsRequest with an additional `.participant` attribute which is the matched ChannelParticipant type for channels/megagroups or ChatParticipants for normal chats.

Example

```python
# Show all user IDs in a chat
for user in client.iter_participants(chat):
    print(user.id)

# Search by name
for user in client.iter_participants(chat, search='name'):
    print(user.username)

# Filter by admins
from telethon.tl.types import ChannelParticipantsAdmins
for user in client.iter_participants(chat,
    filter=ChannelParticipantsAdmins):
    print(user.first_name)
```

iter_profile_photos (*entity: hints.EntityLike*, *limit: int = None*, ***, *offset: int = 0*, *max_id: int = 0*) → telethon.client.chats._ProfilePhotoIter
Iterator over a user's profile photos or a chat's photos.

The order is from the most recent photo to the oldest.

Arguments

entity (`entity`): The entity from which to get the profile or chat photos.

limit (`int` | `None`, optional): Number of photos to be retrieved.

The limit may also be `None`, which would eventually all the photos that are still available.

offset (`int`): How many photos should be skipped before returning the first one.

max_id (`int`): The maximum ID allowed when fetching photos.

Yields Instances of Photo.

Example

```
# Download all the profile photos of some user
for photo in client.iter_profile_photos(user):
    client.download_media(photo)
```

class telethon.client.dialogs.**DialogMethods**

 Bases: `object`

 __dict__ = mappingproxy({'__module__': 'telethon.client.dialogs', 'iter_dialogs': <f

 __weakref__

  list of weak references to the object (if defined)

 conversation(*entity: hints.EntityLike*, ***, *timeout: float = 60*, *total_timeout: float = None*,
   max_messages: int = 100, *exclusive: bool = True*, *replies_are_responses: bool =*
   True) → telethon.tl.custom.conversation.Conversation

  Creates a *Conversation* with the given entity.

  This is not the same as just sending a message to create a "dialog" with them, but rather a way to easily send messages and await for responses or other reactions. Refer to its documentation for more.

  Arguments

   entity (`entity`): The entity with which a new conversation should be opened.

   timeout (`int | float`, optional): The default timeout (in seconds) *per action* to be used. You may also override this timeout on a per-method basis. By default each action can take up to 60 seconds (the value of this timeout).

   total_timeout (`int | float`, optional): The total timeout (in seconds) to use for the whole conversation. This takes priority over per-action timeouts. After these many seconds pass, subsequent actions will result in `asyncio.TimeoutError`.

   max_messages (`int`, optional): The maximum amount of messages this conversation will remember. After these many messages arrive in the specified chat, subsequent actions will result in `ValueError`.

   exclusive (`bool`, optional): By default, conversations are exclusive within a single chat. That means that while a conversation is open in a chat, you can't open another one in the same chat, unless you disable this flag.

    If you try opening an exclusive conversation for a chat where it's already open, it will raise `AlreadyInConversationError`.

   replies_are_responses (`bool`, optional): Whether replies should be treated as responses or not.

    If the setting is enabled, calls to *conv.get_response* and a subsequent call to *conv. get_reply* will return different messages, otherwise they may return the same message.

    Consider the following scenario with one outgoing message, 1, and two incoming messages, the second one replying:

```
              Hello! <1
2> (reply to 1) Hi!
3> (reply to 1) How are you?
```

    And the following code:

```
async with client.conversation(chat) as conv:
    msg1 = await conv.send_message('Hello!')
    msg2 = await conv.get_response()
    msg3 = await conv.get_reply()
```

With the setting enabled, `msg2` will be `'Hi!'` and `msg3` be `'How are you?'` since replies are also responses, and a response was already returned.

With the setting disabled, both `msg2` and `msg3` will be `'Hi!'` since one is a response and also a reply.

Returns A *Conversation*.

Example

```python
# <you> denotes outgoing messages you sent
# <usr> denotes incoming response messages
with bot.conversation(chat) as conv:
    # <you> Hi!
    conv.send_message('Hi!')

    # <usr> Hello!
    hello = conv.get_response()

    # <you> Please tell me your name
    conv.send_message('Please tell me your name')

    # <usr> ?
    name = conv.get_response().raw_text

    while not any(x.isalpha() for x in name):
        # <you> Your name didn't have any letters! Try again
        conv.send_message("Your name didn't have any letters! Try again")

        # <usr> Lonami
        name = conv.get_response().raw_text

    # <you> Thanks Lonami!
    conv.send_message('Thanks {}!'.format(name))
```

delete_dialog(*entity: hints.EntityLike*, ***, *revoke: bool = False*)
Deletes a dialog (leaves a chat or channel).

See also *Dialog.delete()*.

Arguments

> **entity (entities):** The entity of the dialog to delete. If it's a chat or channel, you will leave it. Note that the chat itself is not deleted, only the dialog, because you left it.
>
> **revoke (`bool`, optional):** On private chats, you may revoke the messages from the other peer too. By default, it's `False`. Set it to `True` to delete the history for both.

Returns The Updates object that the request produces, or nothing for private conversations.

Example

```python
# Deleting the first dialog
dialogs = client.get_dialogs(5)
client.delete_dialog(dialogs[0])

# Leaving a channel by username
client.delete_dialog('username')
```

edit_folder(*entity: hints.EntitiesLike = None*, *folder: Union[int, Sequence[int]] = None*, ***, *unpack=None*) → telethon.tl.types.Updates
Edits the folder used by one or more dialogs to archive them.

Arguments

entity (entities): The entity or list of entities to move to the desired archive folder.

folder (`int`): The folder to which the dialog should be archived to.

If you want to "archive" a dialog, use `folder=1`.

If you want to "un-archive" it, use `folder=0`.

You may also pass a list with the same length as `entities` if you want to control where each entity will go.

unpack (`int`, optional): If you want to unpack an archived folder, set this parameter to the folder number that you want to delete.

When you unpack a folder, all the dialogs inside are moved to the folder number 0.

You can only use this parameter if the other two are not set.

Returns The Updates object that the request produces.

Example

```python
# Archiving the first 5 dialogs
dialogs = client.get_dialogs(5)
client.edit_folder(dialogs, 1)

# Un-archiving the third dialog (archiving to folder 0)
client.edit_folder(dialog[2], 0)

# Moving the first dialog to folder 0 and the second to 1
dialogs = client.get_dialogs(2)
client.edit_folder(dialogs, [0, 1])

# Un-archiving all dialogs
client.archive(unpack=1)
```

get_dialogs(**args*, ***kwargs*) → hints.TotalList
Same as *iter_dialogs()*, but returns a *TotalList* instead.

Example

```python
# Get all open conversation, print the title of the first
dialogs = client.get_dialogs()
first = dialogs[0]
print(first.title)

# Use the dialog somewhere else
client.send_message(first, 'hi')

# Getting only non-archived dialogs (both equivalent)
non_archived = client.get_dialogs(folder=0)
non_archived = client.get_dialogs(archived=False)

# Getting only archived dialogs (both equivalent)
archived = client.get_dialogs(folder=1)
non_archived = client.get_dialogs(archived=True)
```

get_drafts() → hints.TotalList
Same as *iter_drafts()*, but returns a list instead.

Example

```
# Get drafts, print the text of the first
drafts = client.get_drafts()
print(drafts[0].text)
```

iter_dialogs(*limit: float = None*, ***, *offset_date: hints.DateLike = None*, *offset_id: int = 0*, *offset_peer: hints.EntityLike = <telethon.tl.types.InputPeerEmpty object>*, *ignore_pinned: bool = False*, *ignore_migrated: bool = False*, *folder: int = None*, *archived: bool = None*) → telethon.client.dialogs._DialogsIter
Iterator over the dialogs (open conversations/subscribed channels).

The order is the same as the one seen in official applications (first pinned, them from those with the most recent message to those with the oldest message).

Arguments

    limit (`int` | `None`): How many dialogs to be retrieved as maximum. Can be set to `None` to retrieve all dialogs. Note that this may take whole minutes if you have hundreds of dialogs, as Telegram will tell the library to slow down through a `FloodWaitError`.

    offset_date (`datetime`, **optional**): The offset date to be used.

    offset_id (`int`, **optional**): The message ID to be used as an offset.

    offset_peer (**InputPeer**, **optional**): The peer to be used as an offset.

    ignore_pinned (`bool`, **optional**): Whether pinned dialogs should be ignored or not. When set to `True`, these won't be yielded at all.

    ignore_migrated (`bool`, **optional**): Whether Chat that have `migrated_to` a Channel should be included or not. By default all the chats in your dialogs are returned, but setting this to `True` will ignore (i.e. skip) them in the same way official applications do.

    folder (`int`, **optional**): The folder from which the dialogs should be retrieved.

        If left unspecified, all dialogs (including those from folders) will be returned.

        If set to `0`, all dialogs that don't belong to any folder will be returned.

        If set to a folder number like `1`, only those from said folder will be returned.

        By default Telegram assigns the folder ID `1` to archived chats, so you should use that if you need to fetch the archived dialogs.

    archived (`bool`, **optional**): Alias for `folder`. If unspecified, all will be returned, `False` implies `folder=0` and `True` implies `folder=1`.

Yields Instances of *Dialog*.

Example

```
# Print all dialog IDs and the title, nicely formatted
for dialog in client.iter_dialogs():
    print('{:>14}: {}'.format(dialog.id, dialog.title))
```

iter_drafts() → telethon.client.dialogs._DraftsIter
Iterator over all open draft messages.

The order is unspecified.

Yields Instances of *Draft*.

Example

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

class telethon.client.downloads.**DownloadMethods**

Bases: object

__dict__ = mappingproxy({'__module__': 'telethon.client.downloads', 'download_profile

__weakref__

list of weak references to the object (if defined)

download_file(*input_location: hints.FileLike*, *file: hints.OutFileLike = None*, ***, *part_size_kb: float = None*, *file_size: int = None*, *progress_callback: hints.ProgressCallback = None*, *dc_id: int = None*) → Optional[bytes]

Low-level method to download files from their input location.

Arguments

> **input_location (InputFileLocation):** The file location from which the file will be downloaded. See *telethon.utils.get_input_location* source for a complete list of supported types.
>
> **file (`str` | `file`, optional):** The output file path, directory, or stream-like object. If the path exists and is a file, it will be overwritten.
>
> > If the file path is None or bytes, then the result will be saved in memory and returned as bytes.
>
> **part_size_kb (`int`, optional):** Chunk size when downloading files. The larger, the less requests will be made (up to 512KB maximum).
>
> **file_size (`int`, optional):** The file size that is about to be downloaded, if known. Only used if progress_callback is specified.
>
> **progress_callback (`callable`, optional):** A callback function accepting two parameters: (downloaded bytes, total). Note that the total is the provided file_size.
>
> **dc_id (`int`, optional):** The data center the library should connect to in order to download the file. You shouldn't worry about this.

Example

```
# Download a file and print its header
data = client.download_file(input_file, bytes)
print(data[:16])
```

download_media(*message: hints.MessageLike*, *file: hints.FileLike = None*, ***, *thumb: hints.FileLike = None*, *progress_callback: hints.ProgressCallback = None*) → Union[str, bytes, None]

Downloads the given media from a message object.

Note that if the download is too slow, you should consider installing cryptg (through pip install cryptg) so that decrypting the received data is done in C instead of Python (much faster).

See also *Message.download_media()*.

Arguments

> **message (*Message* | Media):** The media or message containing the media that will be downloaded.
>
> **file (`str` | `file`, optional):** The output file path, directory, or stream-like object. If the path exists and is a file, it will be overwritten. If file is the type bytes, it will be downloaded in-memory as a bytestring (e.g. file=bytes).

progress_callback (`callable`, optional): A callback function accepting two parameters: `(received bytes, total)`.

thumb (`int` | PhotoSize, optional): Which thumbnail size from the document or photo to download, instead of downloading the document or photo itself.

If it's specified but the file does not have a thumbnail, this method will return `None`.

The parameter should be an integer index between `0` and `len(sizes)`. `0` will download the smallest thumbnail, and `len(sizes) - 1` will download the largest thumbnail. You can also use negative indices.

You can also pass the PhotoSize instance to use.

In short, use `thumb=0` if you want the smallest thumbnail and `thumb=-1` if you want the largest thumbnail.

Returns `None` if no media was provided, or if it was Empty. On success the file path is returned since it may differ from the one given.

Example

```
path = client.download_media(message)
client.download_media(message, filename)
# or
path = message.download_media()
message.download_media(filename)
```

download_profile_photo(*entity: hints.EntityLike*, *file: hints.FileLike = None*, ***, *download_big: bool = True*) → Optional[str]
Downloads the profile photo from the given user, chat or channel.

Arguments

entity (`entity`): From who the photo will be downloaded.

Note: This method expects the full entity (which has the data to download the photo), not an input variant.

It's possible that sometimes you can't fetch the entity from its input (since you can get errors like `ChannelPrivateError`) but you already have it through another call, like getting a forwarded message from it.

---

file (`str` | `file`, optional): The output file path, directory, or stream-like object. If the path exists and is a file, it will be overwritten. If file is the type `bytes`, it will be downloaded in-memory as a bytestring (e.g. `file=bytes`).

download_big (`bool`, optional): Whether to use the big version of the available photos.

Returns `None` if no photo was provided, or if it was Empty. On success the file path is returned since it may differ from the one given.

Example

```
# Download your own profile photo
path = client.download_profile_photo('me')
print(path)
```

iter_download(*file: hints.FileLike*, *, *offset: int = 0*, *stride: int = None*, *limit: int = None*,
chunk_size: int = None, *request_size: int = 524288*, *file_size: int = None*, *dc_id:*
int = None)

Iterates over a file download, yielding chunks of the file.

This method can be used to stream files in a more convenient way, since it offers more control (pausing, resuming, etc.)

Note: Using a value for `offset` or `stride` which is not a multiple of the minimum allowed `request_size`, or if `chunk_size` is different from `request_size`, the library will need to do a bit more work to fetch the data in the way you intend it to.

You normally shouldn't worry about this.

Arguments

> **file (`hints.FileLike`):** The file of which contents you want to iterate over.
>
> **offset (`int`, optional):** The offset in bytes into the file from where the download should start. For example, if a file is 1024KB long and you just want the last 512KB, you would use `offset=512 * 1024`.
>
> **stride (`int`, optional):** The stride of each chunk (how much the offset should advance between reading each chunk). This parameter should only be used for more advanced use cases.
>
> It must be bigger than or equal to the `chunk_size`.
>
> **limit (`int`, optional):** The limit for how many *chunks* will be yielded at most.
>
> **chunk_size (`int`, optional):** The maximum size of the chunks that will be yielded. Note that the last chunk may be less than this value. By default, it equals to `request_size`.
>
> **request_size (`int`, optional):** How many bytes will be requested to Telegram when more data is required. By default, as many bytes as possible are requested. If you would like to request data in smaller sizes, adjust this parameter.
>
> Note that values outside the valid range will be clamped, and the final value will also be a multiple of the minimum allowed size.
>
> **file_size (`int`, optional):** If the file size is known beforehand, you should set this parameter to said value. Depending on the type of the input file passed, this may be set automatically.
>
> **dc_id (`int`, optional):** The data center the library should connect to in order to download the file. You shouldn't worry about this.

Yields

> `bytes` objects representing the chunks of the file if the right conditions are met, or `memoryview` objects instead.

Example

```
# Streaming `media` to an output file
# After the iteration ends, the sender is cleaned up
with open('photo.jpg', 'wb') as fd:
    for chunk client.iter_download(media):
        fd.write(chunk)

# Fetching only the header of a file (32 bytes)
```

(continues on next page)

```python
# You should manually close the iterator in this case.
stream = client.iter_download(media, request_size=32)
header = next(stream)
stream.close()
assert len(header) == 32

# Fetching only the header, inside of an ``async def``
async def main():
    stream = client.iter_download(media, request_size=32)
    header = await stream.__anext__()
    await stream.close()
    assert len(header) == 32
```

class telethon.client.messageparse.**MessageParseMethods**

    Bases: object

    __dict__ = mappingproxy({'__module__': 'telethon.client.messageparse', 'parse_mode':

    __weakref__

        list of weak references to the object (if defined)

    parse_mode

        This property is the default parse mode used when sending messages. Defaults to telethon. extensions.markdown. It will always be either None or an object with parse and unparse methods.

        When setting a different value it should be one of:

            • Object with parse and unparse methods.

            • A callable to act as the parse method.

            • A str indicating the parse_mode. For Markdown 'md' or 'markdown' may be used. For HTML, 'htm' or 'html' may be used.

        The parse method should be a function accepting a single parameter, the text to parse, and returning a tuple consisting of (parsed message str, [MessageEntity instances]).

        The unparse method should be the inverse of parse such that assert text == unparse(*parse(text)).

        See MessageEntity for allowed message entities.

        Example

```python
# Disabling default formatting
client.parse_mode = None

# Enabling HTML as the default format
client.parse_mode = 'html'
```

class telethon.client.messages.**MessageMethods**

    Bases: object

    __dict__ = mappingproxy({'__module__': 'telethon.client.messages', 'iter_messages':

    __weakref__

        list of weak references to the object (if defined)

delete_messages(*entity: hints.EntityLike, message_ids: typing.Union[hints.MessageIDLike, typ-ing.Sequence[hints.MessageIDLike]], *, revoke: bool = True*) → typ-ing.Sequence[types.messages.AffectedMessages]
Deletes the given messages, optionally "for everyone".

See also *Message.delete()*.

> **Warning:** This method does **not** validate that the message IDs belong to the chat that you passed! It's possible for the method to delete messages from different private chats and small group chats at once, so make sure to pass the right IDs.

Arguments

> **entity (`entity`):** From who the message will be deleted. This can actually be `None` for normal chats, but **must** be present for channels and megagroups.
>
> **message_ids (`list` | `int` | *Message*):** The IDs (or ID) or messages to be deleted.
>
> **revoke (`bool`, optional):** Whether the message should be deleted for everyone or not. By default it has the opposite behaviour of official clients, and it will delete the message for everyone.
>
> > Since 24 March 2019, you can also revoke messages of any age (i.e. messages sent long in the past) the *other* person sent in private conversations (and of course your messages too).
> >
> > Disabling this has no effect on channels or megagroups, since it will unconditionally delete the message for everyone.
>
> **Returns** A list of AffectedMessages, each item being the result for the delete calls of the messages in chunks of 100 each.

Example

```
client.delete_messages(chat, messages)
```

edit_message(*entity: typing.Union[hints.EntityLike, types.Message], message: hints.MessageLike = None, text: str = None, *, parse_mode: str = (), link_preview: bool = True, file: hints.FileLike = None, buttons: hints.MarkupLike = None*) → types.Message
Edits the given message to change its text or media.

See also *Message.edit()*.

Arguments

> **entity (`entity` | *Message*):** From which chat to edit the message. This can also be the message to be edited, and the entity will be inferred from it, so the next parameter will be assumed to be the message text.
>
> > You may also pass a InputBotInlineMessageID, which is the only way to edit messages that were sent after the user selects an inline query result.
>
> **message (`int` | *Message* | `str`):** The ID of the message (or *Message* itself) to be edited. If the entity was a *Message*, then this message will be treated as the new text.
>
> **text (`str`, optional):** The new text of the message. Does nothing if the entity was a *Message*.
>
> **parse_mode (`object`, optional):** See the *TelegramClient.parse_mode* property for al-lowed values. Markdown parsing will be used by default.
>
> **link_preview (`bool`, optional):** Should the link preview be shown?

file (`str`|`bytes`|`file`|`media`, optional): The file object that should replace the existing media in the message.

buttons (`list`, *`custom.Button`*, KeyboardButton): The matrix (list of lists), row list or button to be shown after sending the message. This parameter will only work if you have signed in as a bot. You can also pass your own ReplyMarkup here.

Returns The edited *Message*, unless `entity` was a InputBotInlineMessageID in which case this method returns a boolean.

Raises `MessageAuthorRequiredError` if you're not the author of the message but tried editing it anyway.

`MessageNotModifiedError` if the contents of the message were not modified at all.

Example

```
message = client.send_message(chat, 'hello')

client.edit_message(chat, message, 'hello!')
# or
client.edit_message(chat, message.id, 'hello!!')
# or
client.edit_message(message, 'hello!!!')
```

forward_messages(*entity: hints.EntityLike, messages: typing.Union[hints.MessageIDLike, typing.Sequence[hints.MessageIDLike]], from_peer: hints.EntityLike = None, *, silent: bool = None, as_album: bool = None*) → typing.Sequence[types.Message]
Forwards the given messages to the specified entity.

If you want to "forward" a message without the forward header (the "forwarded from" text), you should use *send_message* with the original message instead. This will send a copy of it.

See also *Message.forward_to()*.

Arguments

entity (`entity`): To which entity the message(s) will be forwarded.

messages (`list`|`int`|*`Message`*): The message(s) to forward, or their integer IDs.

from_peer (`entity`): If the given messages are integer IDs and not instances of the `Message` class, this *must* be specified in order for the forward to work. This parameter indicates the entity from which the messages should be forwarded.

silent (`bool`, optional): Whether the message should notify people in a broadcast channel or not. Defaults to `False`, which means it will notify them. Set it to `True` to alter this behaviour.

as_album (`bool`, optional): Whether several image messages should be forwarded as an album (grouped) or not. The default behaviour is to treat albums specially and send outgoing requests with `as_album=True` only for the albums if message objects are used. If IDs are used it will group by default.

In short, the default should do what you expect, `True` will group always (even converting separate images into albums), and `False` will never group.

Returns The list of forwarded *Message*, or a single one if a list wasn't provided as input.

Note that if all messages are invalid (i.e. deleted) the call will fail with `MessageIdInvalidError`. If only some are invalid, the list will have `None` instead of those messages.

Example

```python
# a single one
client.forward_messages(chat, message)
# or
client.forward_messages(chat, message_id, from_chat)
# or
message.forward_to(chat)

# multiple
client.forward_messages(chat, messages)
# or
client.forward_messages(chat, message_ids, from_chat)

# Forwarding as a copy
client.send_message(chat, message)
```

get_messages(**args*, ***kwargs*) → hints.TotalList
> Same as *iter_messages()*, but returns a *TotalList* instead.

> If the limit is not set, it will be 1 by default unless both min_id **and** max_id are set (as *named* arguments), in which case the entire range will be returned.

> This is so because any integer limit would be rather arbitrary and it's common to only want to fetch one message, but if a range is specified it makes sense that it should return the entirety of it.

> If ids is present in the *named* arguments and is not a list, a single *Message* will be returned for convenience instead of a list.

> **Example**

```python
# Get 0 photos and print the total to show how many photos there are
from telethon.tl.types import InputMessagesFilterPhotos
photos = client.get_messages(chat, 0, filter=InputMessagesFilterPhotos)
print(photos.total)

# Get all the photos
photos = client.get_messages(chat, None, filter=InputMessagesFilterPhotos)

# Get messages by ID:
message_1337 = client.get_messages(chats, ids=1337)
```

iter_messages(*entity: hints.EntityLike*, *limit: float = None*, ***, *offset_date: hints.DateLike = None*, *offset_id: int = 0*, *max_id: int = 0*, *min_id: int = 0*, *add_offset: int = 0*, *search: str = None*, *filter: typing.Union[types.TypeMessagesFilter, typing.Type[types.TypeMessagesFilter]] = None*, *from_user: hints.EntityLike = None*, *wait_time: float = None*, *ids: typing.Union[int, typing.Sequence[int]] = None*, *reverse: bool = False*) → typing.Union[_MessagesIter, _IDsIter]
> Iterator over the messages for the given chat.

> The default order is from newest to oldest, but this behaviour can be changed with the reverse parameter.

> If either search, filter or from_user are provided, messages.Search will be used instead of messages.getHistory.

> **Note:** Telegram's flood wait limit for GetHistoryRequest seems to be around 30 seconds per 10 requests, therefore a sleep of 1 second is the default for this limit (or above).

Arguments

entity (`entity`): The entity from whom to retrieve the message history.

It may be `None` to perform a global search, or to get messages by their ID from no particular chat. Note that some of the offsets will not work if this is the case.

Note that if you want to perform a global search, you **must** set a non-empty `search` string.

limit (`int` | `None`, optional): Number of messages to be retrieved. Due to limitations with the API retrieving more than 3000 messages will take longer than half a minute (or even more based on previous calls).

The limit may also be `None`, which would eventually return the whole history.

offset_date (`datetime`): Offset date (messages *previous* to this date will be retrieved). Exclusive.

offset_id (`int`): Offset message ID (only messages *previous* to the given ID will be retrieved). Exclusive.

max_id (`int`): All the messages with a higher (newer) ID or equal to this will be excluded.

min_id (`int`): All the messages with a lower (older) ID or equal to this will be excluded.

add_offset (`int`): Additional message offset (all of the specified offsets + this offset = older messages).

search (`str`): The string to be used as a search query.

filter (MessagesFilter | `type`): The filter to use when returning messages. For instance, InputMessagesFilterPhotos would yield only messages containing photos.

from_user (`entity`): Only messages from this user will be returned. This parameter will be ignored if it is not an user.

wait_time (`int`): Wait time (in seconds) between different GetHistoryRequest. Use this parameter to avoid hitting the `FloodWaitError` as needed. If left to `None`, it will default to 1 second only if the limit is higher than 3000.

ids (`int`, `list`): A single integer ID (or several IDs) for the message that should be returned. This parameter takes precedence over the rest (which will be ignored if this is set). This can for instance be used to get the message with ID 123 from a channel. Note that if the message doesn't exist, `None` will appear in its place, so that zipping the list of IDs with the messages can match one-to-one.

Note: At the time of writing, Telegram will **not** return MessageEmpty for InputMessageReplyTo IDs that failed (i.e. the message is not replying to any, or is replying to a deleted message). This means that it is **not** possible to match messages one-by-one, so be careful if you use non-integers in this parameter.

---

reverse (`bool`, optional): If set to `True`, the messages will be returned in reverse order (from oldest to newest, instead of the default newest to oldest). This also means that the meaning of `offset_id` and `offset_date` parameters is reversed, although they will still be exclusive. `min_id` becomes equivalent to `offset_id` instead of being `max_id` as well since messages are returned in ascending order.

You cannot use this if both `entity` and `ids` are `None`.

Yields Instances of *`Message`*.

Example

```
# From most-recent to oldest
for message in client.iter_messages(chat):
    print(message.id, message.text)

# From oldest to most-recent
for message in client.iter_messages(chat, reverse=True):
    print(message.id, message.text)

# Filter by sender
for message in client.iter_messages(chat, from_user='me'):
    print(message.text)

# Server-side search with fuzzy text
for message in client.iter_messages(chat, search='hello'):
    print(message.id)

# Filter by message type:
from telethon.tl.types import InputMessagesFilterPhotos
for message in client.iter_messages(chat,
→filter=InputMessagesFilterPhotos):
    print(message.photo)
```

pin_message(*entity: hints.EntityLike, message: typing.Optional[hints.MessageIDLike], *, notify: bool = False*)
Pins or unpins a message in a chat.

The default behaviour is to *not* notify members, unlike the official applications.

See also *Message.pin()*.

Arguments

entity (`entity`): The chat where the message should be pinned.

message (`int` | *Message*): The message or the message ID to pin. If it's `None`, the message will be unpinned instead.

notify (`bool`, optional): Whether the pin should notify people or not.

Example

```
# Send and pin a message to annoy everyone
message = client.send_message(chat, 'Pinotifying is fun!')
client.pin_message(chat, message, notify=True)
```

send_message(*entity: hints.EntityLike, message: hints.MessageLike = '', *, reply_to: typing.Union[int, types.Message] = None, parse_mode: Optional[str] = (), link_preview: bool = True, file: hints.FileLike = None, force_document: bool = False, clear_draft: bool = False, buttons: hints.MarkupLike = None, silent: bool = None*) → types.Message*
Sends a message to the specified user, chat or channel.

The default parse mode is the same as the official applications (a custom flavour of mark-down). `**bold**`, `` `code` `` or `__italic__` are available. In addition you can send `[links](https://example.com)` and `[mentions](@username)` (or using IDs like in the Bot API: `[mention](tg://user?id=123456789)`) and `pre` blocks with three backticks.

Sending a `/start` command with a parameter (like `?start=data`) is also done through this method. Simply send `'/start data'` to the bot.

See also *Message.respond()* and *Message.reply()*.

Arguments

> **entity (`entity`):** To who will it be sent.
>
> **message (`str` | *Message*):** The message to be sent, or another message object to resend.
>
> > The maximum length for a message is 35,000 bytes or 4,096 characters. Longer messages will not be sliced automatically, and you should slice them manually if the text to send is longer than said length.
>
> **reply_to (`int` | *Message*, optional):** Whether to reply to a message or not. If an integer is provided, it should be the ID of the message that it should reply to.
>
> **parse_mode (`object`, optional):** See the *TelegramClient.parse_mode* property for allowed values. Markdown parsing will be used by default.
>
> **link_preview (`bool`, optional):** Should the link preview be shown?
>
> **file (`file`, optional):** Sends a message with a file attached (e.g. a photo, video, audio or document). The `message` may be empty.
>
> **force_document (`bool`, optional):** Whether to send the given file as a document or not.
>
> **clear_draft (`bool`, optional):** Whether the existing draft should be cleared or not. Has no effect when sending a file.
>
> **buttons (`list`, *custom.Button*, KeyboardButton):** The matrix (list of lists), row list or button to be shown after sending the message. This parameter will only work if you have signed in as a bot. You can also pass your own ReplyMarkup here.
>
> > All the following limits apply together:
> >
> > - There can be 100 buttons at most (any more are ignored).
> >
> > - There can be 8 buttons per row at most (more are ignored).
> >
> > - The maximum callback data per button is 64 bytes.
> >
> > - The maximum data that can be embedded in total is just over 4KB, shared between inline callback data and text.
>
> **silent (`bool`, optional):** Whether the message should notify people in a broadcast channel or not. Defaults to `False`, which means it will notify them. Set it to `True` to alter this behaviour.

Returns The sent *custom.Message*.

Example

```python
# Markdown is the default
client.send_message('lonami', 'Thanks for the **Telethon** library!')

# Default to another parse mode
client.parse_mode = 'html'

client.send_message('me', 'Some <b>bold</b> and <i>italic</i> text')
client.send_message('me', 'An <a href="https://example.com">URL</a>')
# code and pre tags also work, but those break the documentation :)
client.send_message('me', '<a href="tg://user?id=me">Mentions</a>')

# Explicit parse mode
# No parse mode by default
client.parse_mode = None
```

(continues on next page)

```python
# ...but here I want markdown
client.send_message('me', 'Hello, **world**!', parse_mode='md')

# ...and here I need HTML
client.send_message('me', 'Hello, <i>world</i>!', parse_mode='html')

# If you logged in as a bot account, you can send buttons
from telethon import events, Button

@client.on(events.CallbackQuery)
async def callback(event):
    await event.edit('Thank you for clicking {}!'.format(event.data))

# Single inline button
client.send_message(chat, 'A single button, with "clk1" as data',
                    buttons=Button.inline('Click me', b'clk1'))

# Matrix of inline buttons
client.send_message(chat, 'Pick one from this grid', buttons=[
    [Button.inline('Left'), Button.inline('Right')],
    [Button.url('Check this site!', 'https://lonamiwebs.github.io')]
])

# Reply keyboard
client.send_message(chat, 'Welcome', buttons=[
    Button.text('Thanks!', resize=True, single_use=True),
    Button.request_phone('Send phone'),
    Button.request_location('Send location')
])

# Forcing replies or clearing buttons.
client.send_message(chat, 'Reply to me', buttons=Button.force_reply())
client.send_message(chat, 'Bye Keyboard!', buttons=Button.clear())
```

send_read_acknowledge(*entity: hints.EntityLike*, *message: typing.Union[hints.MessageIDLike*, *typing.Sequence[hints.MessageIDLike]] = None*, ***, *max_id: int = None*, *clear_mentions: bool = False*) → bool

Marks messages as read and optionally clears mentions.

This effectively marks a message as read (or more than one) in the given conversation.

If neither message nor maximum ID are provided, all messages will be marked as read by assuming that `max_id = 0`.

Arguments

> **entity (`entity`):** The chat where these messages are located.
>
> **message (`list` | *`Message`*):** Either a list of messages or a single message.
>
> **max_id (`int`):** Overrides messages, until which message should the acknowledge should be sent.
>
> **clear_mentions (`bool`):** Whether the mention badge should be cleared (so that there are no more mentions) or not for the given entity.
>
> > If no message is provided, this will be the only action taken.

Example

```
client.send_read_acknowledge(last_message)
# or
client.send_read_acknowledge(last_message_id)
# or
client.send_read_acknowledge(messages)
```

class telethon.client.updates.**EventBuilderDict**(*client: TelegramClient*, *update*, *others*)
    Bases: object

    Helper "dictionary" to return events from types and cache them.

    __dict__ = mappingproxy({'__module__': 'telethon.client.updates', '__doc__': '\n Help

    __getitem__(*builder*)

    __weakref__
        list of weak references to the object (if defined)

class telethon.client.updates.**UpdateMethods**
    Bases: object

    __dict__ = mappingproxy({'__module__': 'telethon.client.updates', '_run_until_disconne

    __weakref__
        list of weak references to the object (if defined)

    add_event_handler(*callback: callable*, *event: telethon.events.common.EventBuilder = None*)
        Registers a new event handler callback.

        The callback will be called when the specified event occurs.

        Arguments

            callback (`callable`): The callable function accepting one parameter to be used.

                Note that if you have used *telethon.events.register* in the callback, event will be ignored, and instead the events you previously registered will be used.

            event (_EventBuilder|`type`, optional): The event builder class or instance to be used, for instance events.NewMessage.

                If left unspecified, *telethon.events.raw.Raw* (the Update objects with no further processing) will be passed instead.

        Example

```
from telethon import TelegramClient, events
client = TelegramClient(...)

async def handler(event):
    ...

client.add_event_handler(handler, events.NewMessage)
```

    catch_up()
        "Catches up" on the missed updates while the client was offline. You should call this method after registering the event handlers so that the updates it loads can by processed by your script.

        This can also be used to forcibly fetch new updates if there are any.

        Example

```
client.catch_up()
```

list_event_handlers() → typing.Sequence[typing.Tuple[callable, EventBuilder]]
Lists all registered event handlers.

Returns A list of pairs consisting of (callback, event).

Example

```python
@client.on(events.NewMessage(pattern='hello'))
async def on_greeting(event):
    '''Greets someone'''
    await event.reply('Hi')

for callback, event in client.list_event_handlers():
    print(id(callback), type(event))
```

on(*event: telethon.events.common.EventBuilder*)
Decorator used to *add_event_handler* more conveniently.

Arguments

event (_EventBuilder | type): The event builder class or instance to be used, for instance
events.NewMessage.

Example

```python
from telethon import TelegramClient, events
client = TelegramClient(...)

# Here we use client.on
@client.on(events.NewMessage)
async def handler(event):
    ...
```

remove_event_handler(*callback: callable*, *event: telethon.events.common.EventBuilder = None*)
→ int
Inverse operation of *add_event_handler()*.

If no event is given, all events for this callback are removed. Returns how many callbacks were removed.

Example

```python
@client.on(events.Raw)
@client.on(events.NewMessage)
async def handler(event):
    ...

# Removes only the "Raw" handling
# "handler" will still receive "events.NewMessage"
client.remove_event_handler(handler, events.Raw)

# "handler" will stop receiving anything
client.remove_event_handler(handler)
```

run_until_disconnected()
Runs the event loop until the library is disconnected.

It also notifies Telegram that we want to receive updates as described in https://core.telegram.org/api/
updates.

Manual disconnections can be made by calling *disconnect()* or sending a KeyboardInterrupt (e.g. by pressing Ctrl+C on the console window running the script).

If a disconnection error occurs (i.e. the library fails to reconnect automatically), said error will be raised through here, so you have a chance to except it on your own code.

If the loop is already running, this method returns a coroutine that you should await on your own code.

Note: If you want to handle KeyboardInterrupt in your code, simply run the event loop in your code too in any way, such as loop.run_forever() or await client.disconnected (e.g. loop.run_until_complete(client.disconnected)).

Example

```
# Blocks the current task here until a disconnection occurs.
#
# You will still receive updates, since this prevents the
# script from exiting.
client.run_until_disconnected()
```

class telethon.client.uploads.**UploadMethods**
    Bases: object

    __dict__ = mappingproxy({'__module__': 'telethon.client.uploads', 'send_file': <func

    __weakref__
        list of weak references to the object (if defined)

    send_file(*entity:*   *hints.EntityLike,*   *file:*   *typing.Union[hints.FileLike,*   *typing.Sequence[hints.FileLike]],*   **,*   *caption:*   *Union[str,*   *Sequence[str]]*   *=*   *None,* *force_document:*   *bool*   *=*   *False,*   *progress_callback:*   *hints.ProgressCallback* *=*   *None,*   *reply_to:*   *hints.MessageIDLike*   *=*   *None,*   *attributes:*   *typing.Sequence[types.TypeDocumentAttribute] = None, thumb: hints.FileLike = None,* *allow_cache: bool = True, parse_mode: str = (), voice_note: bool = False, video_note:* *bool = False, buttons: hints.MarkupLike = None, silent: bool = None, supports_streaming:* *bool = False, **kwargs*) → types.Message
    Sends message with the given file to the specified entity.

Note: If the hachoir3 package (hachoir module) is installed, it will be used to determine metadata from audio and video files.

If the pillow package is installed and you are sending a photo, it will be resized to fit within the maximum dimensions allowed by Telegram to avoid errors.PhotoInvalidDimensionsError. This cannot be done if you are sending InputFile, however.

Arguments

    entity (entity): Who will receive the file.

    file (str | bytes | file | media): The file to send, which can be one of:

        • A local file path to an in-disk file. The file name will be the path's base name.

        • A bytes byte array with the file's data to send (for example, by using text. encode('utf-8')). A default file name will be used.

- A bytes `io.IOBase` stream over the file to send (for example, by using `open(file, 'rb')`). Its `.name` property will be used for the file name, or a default if it doesn't have one.

- An external URL to a file over the internet. This will send the file as "external" media, and Telegram is the one that will fetch the media and send it.

- A Bot API-like `file_id`. You can convert previously sent media to file IDs for later reusing with *telethon.utils.pack_bot_file_id*.

- A handle to an existing file (for example, if you sent a message with media before, you can use its `message.media` as a file here).

- A handle to an uploaded file (from *upload_file*).

To send an album, you should provide a list in this parameter.

If a list or similar is provided, the files in it will be sent as an album in the order in which they appear, sliced in chunks of 10 if more than 10 are given.

caption (`str`, optional): Optional caption for the sent media message. When sending an album, the caption may be a list of strings, which will be assigned to the files pairwise.

force_document (`bool`, optional): If left to `False` and the file is a path that ends with the extension of an image file or a video file, it will be sent as such. Otherwise always as a document.

progress_callback (`callable`, optional): A callback function accepting two parameters: `(sent bytes, total)`.

reply_to (`int` | *Message*): Same as `reply_to` from `send_message`.

attributes (`list`, optional): Optional attributes that override the inferred ones, like DocumentAttributeFilename and so on.

thumb (`str` | `bytes` | **file, optional):** Optional JPEG thumbnail (for documents). **Telegram will ignore this parameter** unless you pass a `.jpg` file!

The file must also be small in dimensions and in-disk size. Successful thumbnails were files below 20kb and 200x200px. Width/height and dimensions/size ratios may be important.

allow_cache (`bool`, optional): Whether to allow using the cached version stored in the database or not. Defaults to `True` to avoid re-uploads. Must be `False` if you wish to use different attributes or thumb than those that were used when the file was cached.

parse_mode (`object`, optional): See the *TelegramClient.parse_mode* property for allowed values. Markdown parsing will be used by default.

voice_note (`bool`, optional): If `True` the audio will be sent as a voice note.

Set `allow_cache` to `False` if you sent the same file without this setting before for it to work.

video_note (`bool`, optional): If `True` the video will be sent as a video note, also known as a round video message.

Set `allow_cache` to `False` if you sent the same file without this setting before for it to work.

buttons (`list`, *custom.Button*, KeyboardButton): The matrix (list of lists), row list or button to be shown after sending the message. This parameter will only work if you have signed in as a bot. You can also pass your own ReplyMarkup here.

silent (`bool`, optional): Whether the message should notify people in a broadcast channel or not. Defaults to `False`, which means it will notify them. Set it to `True` to alter this behaviour.

supports_streaming (`bool`, optional): Whether the sent video supports streaming or not. Note that Telegram only recognizes as streamable some formats like MP4, and others like AVI or MKV will

not work. You should convert these to MP4 before sending if you want them to be streamable. Unsupported formats will result in `VideoContentTypeError`.

Returns The *Message* (or messages) containing the sent file, or messages if a list of them was passed.

Example

```python
# Normal files like photos
client.send_file(chat, '/my/photos/me.jpg', caption="It's me!")
# or
client.send_message(chat, "It's me!", file='/my/photos/me.jpg')

# Voice notes or round videos
client.send_file(chat, '/my/songs/song.mp3', voice_note=True)
client.send_file(chat, '/my/videos/video.mp4', video_note=True)

# Custom thumbnails
client.send_file(chat, '/my/documents/doc.txt', thumb='photo.jpg')

# Only documents
client.send_file(chat, '/my/photos/photo.png', force_document=True)

# Albums
client.send_file(chat, [
    '/my/photos/holiday1.jpg',
    '/my/photos/holiday2.jpg',
    '/my/drawings/portrait.png'
])
```

upload_file(*file: hints.FileLike*, ***, *part_size_kb: float = None*, *file_name: str = None*, *use_cache: type = None*, *progress_callback: hints.ProgressCallback = None*) → types.TypeInputFile
Uploads a file to Telegram's servers, without sending it.

This method returns a handle (an instance of InputFile or InputFileBig, as required) which can be later used before it expires (they are usable during less than a day).

Uploading a file will simply return a "handle" to the file stored remotely in the Telegram servers, which can be later used on. This will **not** upload the file to your own chat or any chat at all.

Arguments

file (`str` | `bytes` | `file`): The path of the file, byte array, or stream that will be sent. Note that if a byte array or a stream is given, a filename or its type won't be inferred, and it will be sent as an "unnamed application/octet-stream".

part_size_kb (`int`, optional): Chunk size when uploading files. The larger, the less requests will be made (up to 512KB maximum).

file_name (`str`, optional): The file name which will be used on the resulting InputFile. If not specified, the name will be taken from the `file` and if this is not a `str`, it will be `"unnamed"`.

use_cache (`type`, optional): The type of cache to use (currently either InputDocument or InputPhoto). If present and the file is small enough to need the MD5, it will be checked against the database, and if a match is found, the upload won't be made. Instead, an instance of type `use_cache` will be returned.

progress_callback (`callable`, optional): A callback function accepting two parameters: (sent bytes, total).

> **Returns** [InputFileBig](#) if the file size is larger than 10MB, `InputSizedFile` (subclass of [InputFile](#)) otherwise.
>
> **Example**
>
> ```python
> # Photos as photo and document
> file = client.upload_file('photo.jpg')
> client.send_file(chat, file)                         # sends as photo
> client.send_file(chat, file, force_document=True)  # sends as document
>
> file.name = 'not a photo.jpg'
> client.send_file(chat, file, force_document=True)  # document, new name
>
> # As song or as voice note
> file = client.upload_file('song.ogg')
> client.send_file(chat, file)                         # sends as song
> client.send_file(chat, file, voice_note=True)  # sends as voice note
> ```

class telethon.client.users.**UserMethods**

> Bases: `object`

__call__(*request*, *ordered=False*)
> Call self as a function.

__dict__ = mappingproxy({'__module__': 'telethon.client.users', '__call__': <functio

__weakref__
> list of weak references to the object (if defined)

get_entity(*entity: hints.EntitiesLike*) → hints.Entity
> Turns the given entity into a valid Telegram [User](#), [Chat](#) or [Channel](#). You can also pass a list or iterable of entities, and they will be efficiently fetched from the network.
>
> **Arguments**
>
> > **entity** (`str` | `int` | [Peer](#) | [InputPeer](#)): If a username is given, **the username will be resolved** making an API call every time. Resolving usernames is an expensive operation and will start hitting flood waits around 50 usernames in a short period of time.
> >
> > If you want to get the entity for a *cached* username, you should first `get_input_entity(username)` which will use the cache), and then use `get_entity` with the result of the previous call.
> >
> > Similar limits apply to invite links, and you should use their ID instead.
> >
> > Using phone numbers (from people in your contact list), exact names, integer IDs or [Peer](#) rely on a `get_input_entity` first, which in turn needs the entity to be in cache, unless a [InputPeer](#) was passed.
> >
> > Unsupported types will raise `TypeError`.
> >
> > If the entity can't be found, `ValueError` will be raised.
>
> **Returns** [User](#), [Chat](#) or [Channel](#) corresponding to the input entity. A list will be returned if more than one was given.
>
> **Example**
>
> ```python
> from telethon import utils
>
> me = client.get_entity('me')
> print(utils.get_display_name(me))
> ```

```python
chat = client.get_input_entity('username')
for message in client.iter_messages(chat):
    ...

# Note that you could have used the username directly, but it's
# good to use get_input_entity if you will reuse it a lot.
for message in client.iter_messages('username'):
    ...

# Note that for this to work the phone number must be in your contacts
some_id = client.get_peer_id('+34123456789')
```

get_input_entity(*peer: hints.EntityLike*) → types.TypeInputPeer

Turns the given entity into its input entity version.

Most requests use this kind of InputPeer, so this is the most suitable call to make for those cases. **Generally you should let the library do its job** and don't worry about getting the input entity first, but if you're going to use an entity often, consider making the call:

Arguments

> **entity** (**str** | **int** | **Peer** | **InputPeer**): If a username or invite link is given, **the library will use the cache**. This means that it's possible to be using a username that *changed* or an old invite link (this only happens if an invite link for a small group chat is used after it was upgraded to a mega-group).
>
> If the username or ID from the invite link is not found in the cache, it will be fetched. The same rules apply to phone numbers (`'+34 123456789'`) from people in your contact list.
>
> If an exact name is given, it must be in the cache too. This is not reliable as different people can share the same name and which entity is returned is arbitrary, and should be used only for quick tests.
>
> If a positive integer ID is given, the entity will be searched in cached users, chats or channels, without making any call.
>
> If a negative integer ID is given, the entity will be searched exactly as either a chat (prefixed with `-`) or as a channel (prefixed with `-100`).
>
> If a Peer is given, it will be searched exactly in the cache as either a user, chat or channel.
>
> If the given object can be turned into an input entity directly, said operation will be done.
>
> Unsupported types will raise `TypeError`.
>
> If the entity can't be found, `ValueError` will be raised.

Returns InputPeerUser, InputPeerChat or InputPeerChannel or InputPeerSelf if the parameter is `'me'` or `'self'`.

> If you need to get the ID of yourself, you should use *get_me* with `input_peer=True`) instead.

Example

```python
# If you're going to use "username" often in your code
# (make a lot of calls), consider getting its input entity
# once, and then using the "user" everywhere instead.
user = client.get_input_entity('username')

# The same applies to IDs, chats or channels.
chat = client.get_input_entity(-123456789)
```

get_me (*input_peer: bool = False*) → typing.Union[types.User, types.InputPeerUser]
Gets "me", the current User who is logged in.

If the user has not logged in yet, this method returns `None`.

Arguments

> **input_peer (`bool`, optional):** Whether to return the InputPeerUser version or the normal User. This can be useful if you just need to know the ID of yourself.

Returns Your own User.

Example

```python
print(client.get_me().username)
```

get_peer_id (*peer: hints.EntityLike, add_mark: bool = True*) → int
Gets the ID for the given entity.

This method needs to be `async` because `peer` supports usernames, invite-links, phone numbers (from people in your contact list), etc.

If `add_mark is False`, then a positive ID will be returned instead. By default, bot-API style IDs (signed) are returned.

Example

```python
print(client.get_peer_id('me'))
```

is_bot () → bool
Return `True` if the signed-in user is a bot, `False` otherwise.

Example

```python
if client.is_bot():
    print('Beep')
else:
    print('Hello')
```

is_user_authorized () → bool
Returns `True` if the user is authorized (i.e. has logged in).

Example

```python
if not client.is_user_authorized():
    client.send_code_request(phone)
    code = input('enter code: ')
    client.sign_in(phone, code)
```

2.34 Update Events

Every event (builder) subclasses *common.EventBuilder*, so all the methods in it can be used from any event builder/event instance.

class telethon.events.common.**EventBuilder** (*chats=None,* **,* *blacklist_chats=False,* *func=None*)
Bases: `abc.ABC`

The common event builder, with builtin support to filter per chat.

Args:

> **chats (`entity`, optional):** May be one or more entities (username/peer/etc.), preferably IDs. By default, only matching chats will be handled.
>
> **blacklist_chats (`bool`, optional):** Whether to treat the chats as a blacklist instead of as a whitelist (default). This means that every chat will be handled *except* those specified in `chats` which will be ignored if `blacklist_chats=True`.
>
> **func (`callable`, optional):** A callable function that should accept the event as input parameter, and return a value indicating whether the event should be dispatched or not (any truthy value will do, it does not need to be a `bool`). It works like a custom filter:
>
> ```
> @client.on(events.NewMessage(func=lambda e: e.is_private))
> async def handler(event):
>     pass  # code here
> ```

__dict__ = mappingproxy({'__module__': 'telethon.events.common', '__doc__': '\n The

__weakref__
> list of weak references to the object (if defined)

classmethod build (*update*, *others=None*)
> Builds an event for the given update if possible, or returns None.
>
> `others` are the rest of updates that came in the same container as the current `update`.

filter (*event*)
> If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns None, otherwise the event.
>
> The events must have been resolved before this can be called.

resolve (*client*)
> Helper method to allow event builders to be resolved before usage

self_id = None

class telethon.events.common.**EventCommon**(*chat_peer=None*, *msg_id=None*, *broadcast=None*)
> Bases: *telethon.tl.custom.chatgetter.ChatGetter*, abc.ABC
>
> Intermediate class with common things to all events.
>
> Remember that this class implements *ChatGetter* which means you have access to all chat properties and methods.
>
> In addition, you can access the `original_update` field which contains the original Update.

__str__()
> Return str(self).

client
> The telethon.TelegramClient that created this event.

stringify()

to_dict()

telethon.events.common.**name_inner_event**(*cls*)
> Decorator to rename cls.Event 'Event' as 'cls.Event'

class telethon.events.newmessage.**NewMessage**(*chats=None*, ***, *blacklist_chats=False*, *func=None*, *incoming=None*, *outgoing=None*, *from_users=None*, *forwards=None*, *pattern=None*)

Bases: *telethon.events.common.EventBuilder*

Occurs whenever a new text message or a message with media arrives.

Args:

> **incoming (`bool`, optional):** If set to `True`, only **incoming** messages will be handled. Mutually exclusive with `outgoing` (can only set one of either).

> **outgoing (`bool`, optional):** If set to `True`, only **outgoing** messages will be handled. Mutually exclusive with `incoming` (can only set one of either).

> **from_users (`entity`, optional):** Unlike `chats`, this parameter filters the *senders* of the message. That is, only messages *sent by these users* will be handled. Use `chats` if you want private messages with this/these users. `from_users` lets you filter by messages sent by *one or more* users across the desired chats (doesn't need a list).

> **forwards (`bool`, optional):** Whether forwarded messages should be handled or not. By default, both forwarded and normal messages are included. If it's `True` *only* forwards will be handled. If it's `False` only messages that are *not* forwards will be handled.

> **pattern (`str`, `callable`, `Pattern`, optional):** If set, only messages matching this pattern will be handled. You can specify a regex-like string which will be matched against the message, a callable function that returns `True` if a message is acceptable, or a compiled regex pattern.

class Event(*message*)

> Bases: *telethon.events.common.EventCommon*

> Represents the event of a new message. This event can be treated to all effects as a *Message*, so please **refer to its documentation** to know what you can do with this event.

> **Members:**

>> **message (*Message*):** This is the only difference with the received *Message*, and will return the *telethon.tl.custom.message.Message* itself, not the text.

>> See *Message* for the rest of available members and methods.

>> **pattern_match (`obj`):** The resulting object from calling the passed `pattern` function. Here's an example using a string (defaults to regex match):

```
>>> from telethon import TelegramClient, events
>>> client = TelegramClient(...)
>>>
>>> @client.on(events.NewMessage(pattern=r'hi (\w+)!'))
... async def handler(event):
...     # In this case, the result is a ``Match`` object
...     # since the `str` pattern was converted into
...     # the ``re.compile(pattern).match`` function.
...     print('Welcomed', event.pattern_match.group(1))
...
>>>
```

> **__getattr__**(*item*)

> **__setattr__**(*name*, *value*)

>> Implement setattr(self, name, value).

classmethod build(*update*, *others=None*)
> Builds an event for the given update if possible, or returns None.

> `others` are the rest of updates that came in the same container as the current `update`.

filter(*event*)
> If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns [None](), otherwise the event.

> The events must have been resolved before this can be called.

class telethon.events.chataction.**ChatAction**(*chats=None*, *, *blacklist_chats=False*, *func=None*)
Bases: [*telethon.events.common.EventBuilder*]()

Occurs whenever a user joins or leaves a chat, or a message is pinned.

class Event(*where*, *new_pin=None*, *new_photo=None*, *added_by=None*, *kicked_by=None*, *created=None*, *users=None*, *new_title=None*, *unpin=None*)
> Bases: [*telethon.events.common.EventCommon*]()

> Represents the event of a new chat action.

> **Members:**

>> **action_message ([MessageAction]()):** The message invoked by this Chat Action.

>> **new_pin ([bool]()):** `True` if there is a new pin.

>> **new_photo ([bool]()):** `True` if there's a new chat photo (or it was removed).

>> **photo ([Photo](), optional):** The new photo (or [None]() if it was removed).

>> **user_added ([bool]()):** `True` if the user was added by some other.

>> **user_joined ([bool]()):** `True` if the user joined on their own.

>> **user_left ([bool]()):** `True` if the user left on their own.

>> **user_kicked ([bool]()):** `True` if the user was kicked by some other.

>> **created ([bool](), optional):** `True` if this chat was just created.

>> **new_title ([str](), optional):** The new title string for the chat, if applicable.

>> **unpin ([bool]()):** `True` if the existing pin gets unpinned.

> **added_by**
>> The user who added `users`, if applicable ([None]() otherwise).

> **delete**(**args*, ***kwargs*)
>> Deletes the chat action message. You're responsible for checking whether you have the permission to do so, or to except the error otherwise. Shorthand for [*telethon.client.messages.MessageMethods.delete_messages*]() with `entity` and `message_ids` already set.

>> Does nothing if no message action triggered this event.

> **get_added_by**()
>> Returns [*added_by*]() but will make an API call if necessary.

> **get_input_user**()
>> Returns [*input_user*]() but will make an API call if necessary.

> **get_input_users**()
>> Returns [*input_users*]() but will make an API call if necessary.

get_kicked_by()
    Returns *kicked_by* but will make an API call if necessary.

get_pinned_message()
    If new_pin is True, this returns the *Message* object that was pinned.

get_user()
    Returns *user* but will make an API call if necessary.

get_users()
    Returns *users* but will make an API call if necessary.

input_user
    Input version of the self.user property.

input_users
    Input version of the self.users property.

kicked_by
    The user who kicked users, if applicable (None otherwise).

reply(*args*, **kwargs*)
    Replies to the chat action message (as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with both entity and reply_to already set.

    Has the same effect as *respond* if there is no message.

respond(*args*, **kwargs*)
    Responds to the chat action message (not as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with entity already set.

user
    The first user that takes part in this action (e.g. joined).

    Might be None if the information can't be retrieved or there is no user taking part.

user_id
    Returns the marked signed ID of the first user, if any.

user_ids
    Returns the marked signed ID of the users, if any.

users
    A list of users that take part in this action (e.g. joined).

    Might be empty if the information can't be retrieved or there are no users taking part.

classmethod build(*update*, *others=None*)
    Builds an event for the given update if possible, or returns None.

    others are the rest of updates that came in the same container as the current update.

class telethon.events.userupdate.**UserUpdate**(*chats=None*, *, *blacklist_chats=False*, *func=None*)
    Bases: *telethon.events.common.EventBuilder*

    Occurs whenever a user goes online, starts typing, etc.

    class Event(*user_id*, *, *status=None*, *chat_id=None*, *typing=None*)
        Bases: *telethon.events.common.EventCommon*, *telethon.tl.custom.sendergetter.SenderGetter*

        Represents the event of a user update such as gone online, started typing, etc.

        Members:

status ([UserStatus](), optional): The user status if the update is about going online or offline.

> You should check this attribute first before checking any of the seen within properties, since they will all be `None` if the status is not set.

action ([SendMessageAction](), optional): The "typing" action if any the user is performing if any.

> You should check this attribute first before checking any of the typing properties, since they will all be `None` if the action is not set.

audio
> `True` if what's being recorded/uploaded is an audio.

cancel
> `True` if the action was cancelling other actions.

contact
> `True` if what's being uploaded (selected) is a contact.

document
> `True` if what's being uploaded is document.

geo
> `True` if what's being uploaded is a geo.

get_input_user()
> Alias for *get_input_sender*.

get_user()
> Alias for *get_sender*.

input_user
> Alias for *input_sender*.

last_seen
> Exact `datetime.datetime` when the user was last seen if known.

online
> `True` if the user is currently online,

photo
> `True` if what's being uploaded is a photo.

playing
> `True` if the action is playing a game.

recently
> `True` if the user was seen within a day.

recording
> `True` if the action is recording something.

round
> `True` if what's being recorded/uploaded is a round video.

typing
> `True` if the action is typing a message.

until
> The `datetime.datetime` until when the user should appear online.

uploading
> `True` if the action is uploading something.

> **user**
> > Alias for *sender*.

> **user_id**
> > Alias for *sender_id*.

> **video**
> > `True` if what's being recorded/uploaded is an video.

> **within_months**
> > `True` if the user was seen within 30 days.

> **within_weeks**
> > `True` if the user was seen within 7 days.

classmethod build(*update*, *others=None*)
> Builds an event for the given update if possible, or returns None.
>
> `others` are the rest of updates that came in the same container as the current `update`.

class telethon.events.messageedited.**MessageEdited**(*chats=None*, ***, *blacklist_chats=False*, *func=None*, *incoming=None*, *outgoing=None*, *from_users=None*, *forwards=None*, *pattern=None*)

Bases: *telethon.events.newmessage.NewMessage*

Occurs whenever a message is edited. Just like *NewMessage*, you should treat this event as a *Message*.

> **Warning:** On channels, *Message.out* will be `True` if you sent the message originally, **not if you edited it**! This can be dangerous if you run outgoing commands on edits.
>
> Some examples follow:
>
> - You send a message "A", `out is True`.
>
> - You edit "A" to "B", `out is True`.
>
> - Someone else edits "B" to "C", `out is True` (**be careful!**).
>
> - Someone sends "X", `out is False`.
>
> - Someone edits "X" to "Y", `out is False`.
>
> - You edit "Y" to "Z", `out is False`.
>
> Since there are useful cases where you need the right `out` value, the library cannot do anything automatically to help you. Instead, consider using `from_users='me'` (it won't work in broadcast channels at all since the sender is the channel and not you).

> **class Event**(*message*)
> > Bases: telethon.events.newmessage.Event

> **classmethod build**(*update*, *others=None*)
> > Builds an event for the given update if possible, or returns None.
> >
> > `others` are the rest of updates that came in the same container as the current `update`.

class telethon.events.messagedeleted.**MessageDeleted**(*chats=None*, ***, *blacklist_chats=False*, *func=None*)
> Bases: *telethon.events.common.EventBuilder*

Occurs whenever a message is deleted. Note that this event isn't 100% reliable, since Telegram doesn't always notify the clients that a message was deleted.

Important: Telegram **does not** send information about *where* a message was deleted if it occurs in private conversations with other users or in small group chats, because message IDs are *unique* and you can identify the chat with the message ID alone if you saved it previously.

Telethon **does not** save information of where messages occur, so it cannot know in which chat a message was deleted (this will only work in channels, where the channel ID *is* present).

This means that the `chats=` parameter will not work reliably, unless you intend on working with channels and super-groups only.

---

class Event(*deleted_ids*, *peer*)
> Bases: `telethon.events.common.EventCommon`

classmethod build(*update*, *others=None*)
> Builds an event for the given update if possible, or returns None.
>
> `others` are the rest of updates that came in the same container as the current `update`.

class telethon.events.messageread.**MessageRead**(*chats=None*, *, *blacklist_chats=False*, *func=None*, *inbox=False*)

Bases: `telethon.events.common.EventBuilder`

Occurs whenever one or more messages are read in a chat.

Args:

> **inbox (`bool`, optional):** If this argument is `True`, then when you read someone else's messages the event will be fired. By default (`False`) only when messages you sent are read by someone else will fire it.

class Event(*peer=None*, *max_id=None*, *out=False*, *contents=False*, *message_ids=None*)
> Bases: `telethon.events.common.EventCommon`
>
> Represents the event of one or more messages being read.
>
> **Members:**
>
> > **max_id (`int`):** Up to which message ID has been read. Every message with an ID equal or lower to it have been read.
> >
> > **outbox (`bool`):** `True` if someone else has read your messages.
> >
> > **contents (`bool`):** `True` if what was read were the contents of a message. This will be the case when e.g. you play a voice note. It may only be set on `inbox` events.
>
> **__contains__**(*message*)
> > `True` if the message(s) are read message.
>
> **get_messages**()
> > Returns the list of *Message* **which contents'** were read.
> >
> > Use *is_read()* if you need to check whether a message was read instead checking if it's in here.
>
> **inbox**
> > `True` if you have read someone else's messages.
>
> **is_read**(*message*)
> > Returns `True` if the given message (or its ID) has been read.

If a list-like argument is provided, this method will return a list of booleans indicating which messages have been read.

message_ids
> The IDs of the messages **which contents'** were read.
>
> Use `is_read()` if you need to check whether a message was read instead checking if it's in here.

classmethod build(*update*, *others=None*)
> Builds an event for the given update if possible, or returns None.
>
> `others` are the rest of updates that came in the same container as the current `update`.

filter(*event*)
> If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.
>
> The events must have been resolved before this can be called.

class telethon.events.callbackquery.**CallbackQuery**(*chats=None*, ***, *black-list_chats=False*, *func=None*, *data=None*, *pattern=None*)
> Bases: `telethon.events.common.EventBuilder`
>
> Occurs whenever you sign in as a bot and a user clicks one of the inline buttons on your messages.
>
> Note that the `chats` parameter will **not** work with normal IDs or peers if the clicked inline button comes from a "via bot" message. The `chats` parameter also supports checking against the `chat_instance` which should be used for inline callbacks.
>
> **Args:**
>
> > **data (`bytes`, `str`, `callable`, optional):** If set, the inline button payload data must match this data. A UTF-8 string can also be given, a regex or a callable. For instance, to check against 'data_1' and 'data_2' you can use re.compile(b'data_').
> >
> > **pattern (`bytes`, `str`, `callable`, `Pattern`, optional):** If set, only buttons with payload matching this pattern will be handled. You can specify a regex-like string which will be matched against the payload data, a callable function that returns `True` if a the payload data is acceptable, or a compiled regex pattern.
>
> **class Event**(*query*, *peer*, *msg_id*)
> > Bases: `telethon.events.common.EventCommon`, `telethon.tl.custom.sendergetter.SenderGetter`
> >
> > Represents the event of a new callback query.
> >
> > **Members:**
> >
> > > **query ([UpdateBotCallbackQuery](#)):** The original [UpdateBotCallbackQuery](#).
> > >
> > > **data_match (`obj`, optional):** The object returned by the `data=` parameter when creating the event builder, if any. Similar to `pattern_match` for the new message event.
> > >
> > > **pattern_match (`obj`, optional):** Alias for `data_match`.
> >
> > **answer**(*message=None*, *cache_time=0*, ***, *url=None*, *alert=False*)
> > > Answers the callback query (and stops the loading circle).
> > > **Args:**
> > > > **message (`str`, optional):** The toast message to show feedback to the user.
> > > > **cache_time (`int`, optional):** For how long this result should be cached on the user's client. Defaults to 0 for no cache.
> > > > **url (`str`, optional):** The URL to be opened in the user's client. Note that the only valid URLs are those of games your bot has, or alternatively a 't.me/your_bot?start=xyz' parameter.

> **alert (`bool`, optional):** Whether an alert (a pop-up dialog) should be used instead of showing a toast. Defaults to `False`.

chat_instance
> Unique identifier for the chat where the callback occurred. Useful for high scores in games.

data
> Returns the data payload from the original inline button.

delete(**args*, ***kwargs*)
> Deletes the message. Shorthand for *telethon.client.messages.MessageMethods. delete_messages* with `entity` and `message_ids` already set.
>
> If you need to delete more than one message at once, don't use this *delete* method. Use a *telethon.client.telegramclient.TelegramClient* instance directly.
>
> This method also creates a task to *answer* the callback.
>
> This method will likely fail if *via_inline* is `True`.

edit(**args*, ***kwargs*)
> Edits the message. Shorthand for *telethon.client.messages.MessageMethods. edit_message* with the `entity` set to the correct InputBotInlineMessageID.
>
> Returns `True` if the edit was successful.
>
> This method also creates a task to *answer* the callback.
>
> ---
>
> **Note:** This method won't respect the previous message unlike *Message.edit*, since the message object is normally not present.
>
> ---

get_message()
> Returns the message to which the clicked inline button belongs.

id
> Returns the query ID. The user clicking the inline button is the one who generated this random ID.

message_id
> Returns the message ID to which the clicked inline button belongs.

reply(**args*, ***kwargs*)
> Replies to the message (as a reply). Shorthand for *telethon.client.messages. MessageMethods.send_message* with both `entity` and `reply_to` already set.
>
> This method also creates a task to *answer* the callback.
>
> This method will likely fail if *via_inline* is `True`.

respond(**args*, ***kwargs*)
> Responds to the message (not as a reply). Shorthand for *telethon.client.messages. MessageMethods.send_message* with `entity` already set.
>
> This method also creates a task to *answer* the callback.
>
> This method will likely fail if *via_inline* is `True`.

via_inline
> Whether this callback was generated from an inline button sent via an inline query or not. If the bot sent the message itself with buttons, and one of those is clicked, this will be `False`. If a user sent the message coming from an inline query to the bot, and one of those is clicked, this will be `True`.

> If it's `True`, it's likely that the bot is **not** in the chat, so methods like *respond* or *delete* won't work (but *edit* will always work).

classmethod build(*update*, *others=None*)

Builds an event for the given update if possible, or returns None.

`others` are the rest of updates that came in the same container as the current `update`.

filter(*event*)

If the ID of `event._chat_peer` isn't in the chats set (or it is but the set is a blacklist) returns `None`, otherwise the event.

The events must have been resolved before this can be called.

class telethon.events.inlinequery.**InlineQuery**(*users=None*, *,* *blacklist_users=False*, *func=None*, *pattern=None*)

Bases: *telethon.events.common.EventBuilder*

Occurs whenever you sign in as a bot and a user sends an inline query such as `@bot query`.

Args:

> **users (`entity`, optional):** May be one or more entities (username/peer/etc.), preferably IDs. By default, only inline queries from these users will be handled.
>
> **blacklist_users (`bool`, optional):** Whether to treat the users as a blacklist instead of as a whitelist (default). This means that every chat will be handled *except* those specified in `users` which will be ignored if `blacklist_users=True`.
>
> **pattern (`str`, `callable`, `Pattern`, optional):** If set, only queries matching this pattern will be handled. You can specify a regex-like string which will be matched against the message, a callable function that returns `True` if a message is acceptable, or a compiled regex pattern.

class Event(*query*)

Bases: *telethon.events.common.EventCommon*, *telethon.tl.custom.sendergetter.SenderGetter*

Represents the event of a new callback query.

Members:

> **query ([UpdateBotCallbackQuery](#)):** The original [UpdateBotCallbackQuery](#).
>
> > Make sure to access the *text* of the query if that's what you want instead working with this.
>
> **pattern_match (`obj`, optional):** The resulting object from calling the passed `pattern` function, which is `re.compile(...).match` by default.

answer(*results=None*, *cache_time=0*, *,* *gallery=False*, *next_offset=None*, *private=False*, *switch_pm=None*, *switch_pm_param=''*)

Answers the inline query with the given results.

Args:

> **results (`list`, optional):** A list of [InputBotInlineResult](#) to use. You should use *builder* to create these:

```
builder = inline.builder
r1 = builder.article('Be nice', text='Have a nice day')
r2 = builder.article('Be bad', text="I don't like you")
await inline.answer([r1, r2])
```

> You can send up to 50 results as documented in https://core.telegram.org/bots/api#answerinlinequery. Sending more will raise `ResultsTooMuchError`, and you should consider using `next_offset` to paginate them.

cache_time (**int**, optional): For how long this result should be cached on the user's client. Defaults to 0 for no cache.

gallery (**bool**, optional): Whether the results should show as a gallery (grid) or not.

next_offset (**str**, optional): The offset the client will send when the user scrolls the results and it repeats the request.

private (**bool**, optional): Whether the results should be cached by Telegram (not private) or by the user's client (private).

switch_pm (**str**, optional): If set, this text will be shown in the results to allow the user to switch to private messages.

switch_pm_param (**str**, optional): Optional parameter to start the bot with if switch_pm was used.

Example:

```python
@bot.on(events.InlineQuery)
async def handler(event):
    builder = event.builder

    rev_text = event.text[::-1]
    await event.answer([
        builder.article('Reverse text', text=rev_text),
        builder.photo('/path/to/photo.jpg')
    ])
```

builder

Returns a new *InlineBuilder* instance.

geo

If the user location is requested when using inline mode and the user's device is able to send it, this will return the GeoPoint with the position of the user.

id

Returns the unique identifier for the query ID.

offset

The string the user's client used as an offset for the query. This will either be empty or equal to offsets passed to *answer*.

text

Returns the text the user used to make the inline query.

classmethod build(*update*, *others=None*)

Builds an event for the given update if possible, or returns None.

others are the rest of updates that came in the same container as the current update.

filter(*event*)

If the ID of event._chat_peer isn't in the chats set (or it is but the set is a blacklist) returns None, otherwise the event.

The events must have been resolved before this can be called.

class telethon.events.album.**Album**(*chats=None*, *, *blacklist_chats=False*, *func=None*)

Bases: *telethon.events.common.EventBuilder*

Occurs whenever you receive an album. This event only exists to ease dealing with an unknown amount of messages that belong to the same album.

class Event(*messages*)

Bases: *telethon.events.common.EventCommon*, *telethon.tl.custom.sendergetter.SenderGetter*

Represents the event of a new album.

Members:

messages (Sequence[_Message_]): The list of messages belonging to the same album.

__getitem__(*n*)
Access the n'th message in the album.

Equivalent to `event.messages[n]`.

__iter__()
Iterate over the messages in the album.

Equivalent to `iter(self.messages)`.

__len__()
Return the amount of messages in the album.

Equivalent to `len(self.messages)`.

delete(**args*, ***kwargs*)
Deletes the entire album. You're responsible for checking whether you have the permission to do so, or to except the error otherwise. Shorthand for _telethon.client.messages._ _MessageMethods.delete_messages_ with `entity` and `message_ids` already set.

edit(**args*, ***kwargs*)
Edits the first caption or the message, or the first messages' caption if no caption is set, iff it's outgoing. Shorthand for _telethon.client.messages.MessageMethods.edit_message_ with both `entity` and `message` already set.

Returns _None_ if the message was incoming, or the edited `Message` otherwise.

Note: This is different from _client.edit_message_ and **will respect** the previous state of the message. For example, if the message didn't have a link preview, the edit won't add one by default, and you should force it by setting it to _True_ if you want it.

This is generally the most desired and convenient behaviour, and will work for link previews and message buttons.

---

forward
The _Forward_ information for the first message in the album if it was forwarded.

forward_to(**args*, ***kwargs*)
Forwards the entire album. Shorthand for _telethon.client.messages._ _MessageMethods.forward_messages_ with both `messages` and `from_peer` already set.

get_reply_message()
The _Message_ that this album is replying to, or _None_.

The result will be cached after its first use.

grouped_id
The shared `grouped_id` between all the messages.

is_reply
True if the album is a reply to some other message.

Remember that you can access the ID of the message this one is replying to through `reply_to_msg_id`, and the Message object with _get_reply_message()_.

mark_read()
> Marks the entire album as read. Shorthand for *client.send_read_acknowledge()* with both `entity` and `message` already set.

pin(**, notify=False*)
> Pins the first photo in the album. Shorthand for *telethon.client.messages.MessageMethods.pin_message* with both `entity` and `message` already set.

raw_text
> The raw message text of the first photo with a caption, ignoring any formatting.

reply(**args, **kwargs*)
> Replies to the first photo in the album (as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with both `entity` and `reply_to` already set.

respond(**args, **kwargs*)
> Responds to the album (not as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with `entity` already set.

text
> The message text of the first photo with a caption, formatted using the client's default parse mode.

classmethod build(*update, others=None*)
> Builds an event for the given update if possible, or returns None.
>
> `others` are the rest of updates that came in the same container as the current `update`.

class telethon.events.raw.**Raw**(*types=None, *, func=None*)
> Bases: *telethon.events.common.EventBuilder*

Raw events are not actual events. Instead, they are the raw Update object that Telegram sends. You normally shouldn't need these.

Args:

> **types (list | tuple | type, optional):** The type or types that the Update instance must be. Equivalent to `if not isinstance(update, types):  return`.

classmethod build(*update, others=None*)
> Builds an event for the given update if possible, or returns None.
>
> `others` are the rest of updates that came in the same container as the current `update`.

filter(*event*)
> If the ID of event._chat_peer isn't in the chats set (or it is but the set is a blacklist) returns None, otherwise the event.
>
> The events must have been resolved before this can be called.

resolve(*client*)
> Helper method to allow event builders to be resolved before usage

exception telethon.events.**StopPropagation**
> Bases: Exception

If this exception is raised in any of the handlers for a given event, it will stop the execution of all other registered event handlers. It can be seen as the `StopIteration` in a for loop but for events.

Example usage:

```
>>> from telethon import TelegramClient, events
>>> client = TelegramClient(...)
>>>
>>> @client.on(events.NewMessage)
... async def delete(event):
...     await event.delete()
...     # No other event handler will have a chance to handle this event
...     raise StopPropagation
...
>>> @client.on(events.NewMessage)
... async def _(event):
...     # Will never be reached, because it is the second handler
...     pass
```

__weakref__
>   list of weak references to the object (if defined)

telethon.events.**is_handler**(*callback*)
>   Returns `True` if the given callback is an event handler (i.e. you used *register* on it).

telethon.events.**list**(*callback*)
>   Returns a list containing the registered event builders inside the specified callback handler.

telethon.events.**register**(*event=None*)
>   Decorator method to *register* event handlers. This is the client-less *add_event_handler()* variant.

>   Note that this method only registers callbacks as handlers, and does not attach them to any client. This is useful for external modules that don't have access to the client, but still want to define themselves as a handler. Example:

```
>>> from telethon import events
>>> @events.register(events.NewMessage)
... async def handler(event):
...     ...
...
>>> # (somewhere else)
...
>>> from telethon import TelegramClient
>>> client = TelegramClient(...)
>>> client.add_event_handler(handler)
```

>   Remember that you can use this as a non-decorator through `register(event)(callback)`.

>   **Args:**

>   >   **event (_EventBuilder | type):** The event builder class or instance to be used, for instance `events.NewMessage`.

telethon.events.**unregister**(*callback*, *event=None*)
>   Inverse operation of *register* (though not a decorator). Client-less *remove_event_handler* variant. **Note that this won't remove handlers from the client**, because it simply can't, so you would generally use this before adding the handlers to the client.

>   This method is here for symmetry. You will rarely need to unregister events, since you can simply just not add them to any client.

>   If no event is given, all events for this callback are removed. Returns how many callbacks were removed.

2.35 Custom package

The `telethon.tl.custom` package contains custom classes that the library uses in order to make working with Telegram easier. Only those that you are supposed to use will be documented here. You can use undocumented ones at your own risk.

More often than not, you don't need to import these (unless you want type hinting), nor do you need to manually create instances of these classes. They are returned by client methods.

Contents

2.35.1 AdminLogEvent

class telethon.tl.custom.adminlogevent.**AdminLogEvent**(*original*, *entities*)

    Bases: `object`

    Represents a more friendly interface for admin log events.

    Members:

        original (ChannelAdminLogEvent): The original ChannelAdminLogEvent.

        entities (`dict`): A dictionary mapping user IDs to User.

            When *old* and *new* are ChannelParticipant, you can use this dictionary to map the user_id, kicked_by, inviter_id and promoted_by IDs to their User.

        user (User): The user that caused this action (entities[original.user_id]).

        input_user (InputPeerUser): Input variant of user.

    __dict__ = mappingproxy({'__module__': 'telethon.tl.custom.adminlogevent', '__doc__':

__str__()
> Return str(self).

__weakref__
> list of weak references to the object (if defined)

action
> The original ChannelAdminLogEventAction.

changed_about
> Whether the channel's about was changed or not.
>
> If `True`, *old* and *new* will be present as `str`.

changed_admin
> Whether the permissions for an admin in this channel changed or not.
>
> If `True`, *old* and *new* will be present as ChannelParticipant.

changed_default_banned_rights
> Whether the default banned rights were changed or not.
>
> If `True`, *old* and *new* will be present as ChatBannedRights.

changed_hide_history
> Whether hiding the previous message history for new members in the channel was toggled or not.
>
> If `True`, *old* and *new* will be present as `bool`.

changed_invites
> Whether the invites in the channel were toggled or not.
>
> If `True`, *old* and *new* will be present as `bool`.

changed_location
> Whether the location setting of the channel has changed or not.
>
> If `True`, *old* and *new* will be present as ChannelLocation.

changed_message
> Whether a message in this channel was edited or not.
>
> If `True`, *old* and *new* will be present as *Message*.

changed_photo
> Whether the channel's photo was changed or not.
>
> If `True`, *old* and *new* will be present as Photo.

changed_pin
> Whether a new message in this channel was pinned or not.
>
> If `True`, *new* will be present as *Message*.

changed_restrictions
> Whether a message in this channel was edited or not.
>
> If `True`, *old* and *new* will be present as ChannelParticipant.

changed_signatures
> Whether the message signatures in the channel were toggled or not.
>
> If `True`, *old* and *new* will be present as `bool`.

changed_sticker_set
> Whether the channel's sticker set was changed or not.
>
> If `True`, *old* and *new* will be present as InputStickerSet.

changed_title
> Whether the channel's title was changed or not.
>
> If `True`, *old* and *new* will be present as `str`.

changed_username
> Whether the channel's username was changed or not.
>
> If `True`, *old* and *new* will be present as `str`.

date
> The date when this event occured.

deleted_message
> Whether a message in this channel was deleted or not.
>
> If `True`, *old* will be present as *Message*.

id
> The ID of this event.

joined
> Whether `user` joined through the channel's public username or not.

joined_invite
> Whether a new user joined through an invite link to the channel or not.
>
> If `True`, *new* will be present as ChannelParticipant.

left
> Whether `user` left the channel or not.

new
> The new value present in the event.

old
> The old value from the event.

stopped_poll
> Whether a poll was stopped or not.
>
> If `True`, *new* will be present as *Message*.

stringify()

user_id
> The ID of the user that triggered this event.

2.35.2 Button

class telethon.tl.custom.button.**Button**(*button*, ***, *resize*, *single_use*, *selective*)
> Bases: `object`

Note: This class is used to **define** reply markups, e.g. when sending a message or replying to events. When you access *Message.buttons* they are actually *MessageButton*, so you might want to refer to that class instead.

Helper class to allow defining `reply_markup` when sending a message with inline or keyboard buttons.

You should make use of the defined class methods to create button instances instead making them yourself (i.e. don't do `Button(...)` but instead use methods line *Button.inline(...)* etc.

You can use *inline*, *switch_inline* and *url* together to create inline buttons (under the message).

You can use *text*, *request_location* and *request_phone* together to create a reply markup (replaces the user keyboard). You can also configure the aspect of the reply with these.

You **cannot** mix the two type of buttons together, and it will error if you try to do so.

The text for all buttons may be at most 142 characters. If more characters are given, Telegram will cut the text to 128 characters and add the ellipsis (. . .) character as the 129.

__dict__ = mappingproxy({'__module__': 'telethon.tl.custom.button', '__doc__': "\n .

__weakref__
list of weak references to the object (if defined)

static clear()
Clears all keyboard buttons after sending a message with this markup. When used, no other button should be present or it will be ignored.

static force_reply()
Forces a reply to the message with this markup. If used, no other button should be present or it will be ignored.

static inline(*text*, *data=None*)
Creates a new inline button with some payload data in it.

If `data` is omitted, the given *text* will be used as `data`. In any case `data` should be either `bytes` or `str`.

Note that the given `data` must be less or equal to 64 bytes. If more than 64 bytes are passed as data, `ValueError` is raised.

classmethod request_location(*text*, ***, *resize=None*, *single_use=None*, *selective=None*)
Creates a new keyboard button to request the user's location on click.

`resize`, `single_use` and `selective` are documented in *text*.

classmethod request_phone(*text*, ***, *resize=None*, *single_use=None*, *selective=None*)
Creates a new keyboard button to request the user's phone on click.

`resize`, `single_use` and `selective` are documented in *text*.

static switch_inline(*text*, *query=''*, *same_peer=False*)
Creates a new inline button to switch to inline query.

If `query` is given, it will be the default text to be used when making the inline query.

If `same_peer is True` the inline query will directly be set under the currently opened chat. Otherwise, the user will have to select a different dialog to make the query.

classmethod text(*text*, ***, *resize=None*, *single_use=None*, *selective=None*)
Creates a new keyboard button with the given text.

Args:

resize (`bool`): If present, the entire keyboard will be reconfigured to be resized and be smaller if there are not many buttons.

single_use (`bool`): If present, the entire keyboard will be reconfigured to be usable only once before it hides itself.

> > **selective ([bool](#)):** If present, the entire keyboard will be reconfigured to be "selective". The keyboard
> > will be shown only to specific users. It will target users that are @mentioned in the text of the
> > message or to the sender of the message you reply to.

> **static url**(*text*, *url=None*)
>> Creates a new inline button to open the desired URL on click.
>>
>> If no *url* is given, the *text* will be used as said URL instead.
>>
>> You cannot detect that the user clicked this button directly.

2.35.3 ChatGetter

class telethon.tl.custom.chatgetter.**ChatGetter**(*chat_peer=None*, ***, *input_chat=None*,
<div align="right">*chat=None*, *broadcast=None*)</div>

> Bases: [abc.ABC](#)

> Helper base class that introduces the *chat*, *input_chat* and *chat_id* properties and *get_chat* and
> *get_input_chat* methods.

> **__dict__ = mappingproxy({'__module__': 'telethon.tl.custom.chatgetter', '__doc__': '**

> **__weakref__**
>> list of weak references to the object (if defined)

> **chat**
>> Returns the [User](#), [Chat](#) or [Channel](#) where this object belongs to. It may be [None](#) if Telegram didn't send
>> the chat.
>>
>> If you only need the ID, use *chat_id* instead.
>>
>> If you need to call a method which needs this chat, use *input_chat* instead.
>>
>> If you're using *telethon.events*, use *get_chat()* instead.

> **chat_id**
>> Returns the marked chat integer ID. Note that this value **will be different** from to_id for incoming
>> private messages, since the chat *to* which the messages go is to your own person, but the *chat* itself is with
>> the one who sent the message.
>>
>> TL;DR; this gets the ID that you expect.
>>
>> If there is a chat in the object, *chat_id* will *always* be set, which is why you should use it instead of
>> *chat.id*.

> **get_chat**()
>> Returns *chat*, but will make an API call to find the chat unless it's already cached.
>>
>> If you only need the ID, use *chat_id* instead.
>>
>> If you need to call a method which needs this chat, use *get_input_chat()* instead.

> **get_input_chat**()
>> Returns *input_chat*, but will make an API call to find the input chat unless it's already cached.

> **input_chat**
>> This [InputPeer](#) is the input version of the chat where the message was sent. Similarly to *input_sender*,
>> this doesn't have things like username or similar, but still useful in some cases.
>>
>> Note that this might not be available if the library doesn't have enough information available.

> **is_channel**
>> [True](#) if the message was sent on a megagroup or channel.

is_group
> True if the message was sent on a group or megagroup.
>
> Returns `None` if there isn't enough information (e.g. on `events.MessageDeleted`).

is_private
> `True` if the message was sent as a private message.
>
> Returns `None` if there isn't enough information (e.g. on `events.MessageDeleted`).

2.35.4 Conversation

class telethon.tl.custom.conversation.**Conversation**(*client*, *input_chat*, ***, *timeout*, *total_timeout*, *max_messages*, *exclusive*, *replies_are_responses*)

Bases: `telethon.tl.custom.chatgetter.ChatGetter`

Represents a conversation inside an specific chat.

A conversation keeps track of new messages since it was created until its exit and easily lets you query the current state.

If you need a conversation across two or more chats, you should use two conversations and synchronize them as you better see fit.

__aenter__()

__aexit__(*exc_type*, *exc_val*, *exc_tb*)

__enter__()
> Helps to cut boilerplate on async context managers that offer synchronous variants.

__exit__(**args*)

cancel()
> Cancels the current conversation. Pending responses and subsequent calls to get a response will raise `asyncio.CancelledError`.
>
> This method is synchronous and should not be awaited.

cancel_all()
> Calls *cancel* on *all* conversations in this chat.
>
> Note that you should `await` this method, since it's meant to be used outside of a context manager, and it needs to resolve the chat.

get_edit(*message=None*, ***, *timeout=None*)
> Awaits for an edit after the last message to arrive. The arguments are the same as those for *get_response*.

get_reply(*message=None*, ***, *timeout=None*)
> Gets the next message that explicitly replies to a previous one.

get_response(*message=None*, ***, *timeout=None*)
> Gets the next message that responds to a previous one.
>
> **Args:**
>
> > **message (*Message* | int, optional):** The message (or the message ID) for which a response is expected. By default this is the last sent message.
> >
> > **timeout (int | float, optional):** If present, this `timeout` (in seconds) will override the per-action timeout defined for the conversation.

---

mark_read(*message=None*)
> Marks as read the latest received message if `message is None`. Otherwise, marks as read until the given message (or message ID).
>
> This is equivalent to calling `client.send_read_acknowledge`.

send_file(**args*, ***kwargs*)
> Sends a file in the context of this conversation. Shorthand for `telethon.client.uploads.UploadMethods.send_file` with `entity` already set.

send_message(**args*, ***kwargs*)
> Sends a message in the context of this conversation. Shorthand for `telethon.client.messages.MessageMethods.send_message` with `entity` already set.

wait_event(*event*, ***, *timeout=None*)
> Waits for a custom event to occur. Timeouts still apply.
>
> Unless you're certain that your code will run fast enough, generally you should get a "handle" of this special coroutine before acting. Generally, you should do this:

```python
>>> from telethon import TelegramClient, events
>>>
>>> client = TelegramClient(...)
>>>
>>> async def main():
>>>     async with client.conversation(...) as conv:
>>>         response = conv.wait_event(events.NewMessage(incoming=True))
>>>         await conv.send_message('Hi')
>>>         response = await response
```

> This way your event can be registered before acting, since the response may arrive before your event was registered. It depends on your use case since this also means the event can arrive before you send a previous action.

wait_read(*message=None*, ***, *timeout=None*)
> Awaits for the sent message to be marked as read. Note that receiving a response doesn't imply the message was read, and this action will also trigger even without a response.

2.35.5 Dialog

class telethon.tl.custom.dialog.**Dialog**(*client*, *dialog*, *entities*, *messages*)
> Bases: `object`

Custom class that encapsulates a dialog (an open "conversation" with someone, a group or a channel) providing an abstraction to easily access the input version/normal entity/message etc. The library will return instances of this class when calling `get_dialogs()`.

Args:

> **dialog** (`Dialog`): The original `Dialog` instance.
>
> **pinned** (`bool`): Whether this dialog is pinned to the top or not.
>
> **folder_id** (`folder_id`): The folder ID that this dialog belongs to.
>
> **archived** (`bool`): Whether this dialog is archived or not (`folder_id is None`).
>
> **message** (`Message`): The last message sent on this dialog. Note that this member will not be updated when new messages arrive, it's only set on creation of the instance.
>
> **date** (`datetime`): The date of the last message sent on this dialog.

---

entity (**entity**): The entity that belongs to this dialog (user, chat or channel).

input_entity (InputPeer): Input version of the entity.

id (**int**): The marked ID of the entity, which is guaranteed to be unique.

name (**str**): Display name for this dialog. For chats and channels this is their title, and for users it's "First-Name Last-Name".

title (**str**): Alias for `name`.

unread_count (**int**): How many messages are currently unread in this dialog. Note that this value won't update when new messages arrive.

unread_mentions_count (**int**): How many mentions are currently unread in this dialog. Note that this value won't update when new messages arrive.

draft (*Draft*): The draft object in this dialog. It will not be `None`, so you can call `draft. set_message(...)`.

is_user (**bool**): `True` if the `entity` is a User.

is_group (**bool**): `True` if the `entity` is a Chat or a Channel megagroup.

is_channel (**bool**): `True` if the `entity` is a Channel.

__dict__ = mappingproxy({'__module__': 'telethon.tl.custom.dialog', '__doc__': '\n C

__str__()
 Return str(self).

__weakref__
 list of weak references to the object (if defined)

archive(*folder=1*)
 Archives (or un-archives) this dialog.

 Args:

 folder (**int**, **optional**): The folder to which the dialog should be archived to.

 If you want to "un-archive" it, use `folder=0`.

 Returns: The Updates object that the request produces.

 Example:

 ```
 # Archiving
 dialog.archive()

 # Un-archiving
 dialog.archive(0)
 ```

delete(*revoke=False*)
 Deletes the dialog from your dialog list. If you own the channel this won't destroy it, only delete it from the list.

 Shorthand for *telethon.client.dialogs.DialogMethods.delete_dialog* with `entity` already set.

send_message(**args*, ***kwargs*)
 Sends a message to this dialog. This is just a wrapper around `client.send_message(dialog. input_entity, *args, **kwargs)`.

stringify()

to_dict()

2.35.6 Draft

class telethon.tl.custom.draft.**Draft**(*client*, *peer*, *draft*, *entity*)
    Bases: object

    Custom class that encapsulates a draft on the Telegram servers, providing an abstraction to change the message conveniently. The library will return instances of this class when calling get_drafts().

    Args:

        date (datetime): The date of the draft.

        link_preview (bool): Whether the link preview is enabled or not.

        reply_to_msg_id (int): The message ID that the draft will reply to.

    __dict__ = mappingproxy({'__module__': 'telethon.tl.custom.draft', '__doc__': '\n Cu

    __str__()
        Return str(self).

    __weakref__
        list of weak references to the object (if defined)

    delete()
        Deletes this draft, and returns True on success.

    entity
        The entity that belongs to this dialog (user, chat or channel).

    get_entity()
        Returns *entity* but will make an API call if necessary.

    get_input_entity()
        Returns *input_entity* but will make an API call if necessary.

    input_entity
        Input version of the entity.

    is_empty
        Convenience bool to determine if the draft is empty or not.

    raw_text
        The raw (text without formatting) contained in the draft. It will be empty if there is no text (thus draft not set).

    send(*clear=True*, *parse_mode=()*)
        Sends the contents of this draft to the dialog. This is just a wrapper around send_message(dialog. input_entity, *args, **kwargs).

    set_message(*text=None*, *reply_to=0*, *parse_mode=()*, *link_preview=None*)
        Changes the draft message on the Telegram servers. The changes are reflected in this object.

        Parameters

            • **text** (*str*) – New text of the draft. Preserved if left as None.

            • **reply_to** (*int*) – Message ID to reply to. Preserved if left as 0, erased if set to None.

            • **link_preview** (*bool*) – Whether to attach a web page preview. Preserved if left as None.

- **parse_mode** (`str`) – The parse mode to be used for the text.

> **Return bool** `True` on success.

stringify()

text

> The markdown text contained in the draft. It will be empty if there is no text (and hence no draft is set).

to_dict()

2.35.7 File

class telethon.tl.custom.file.**File**(*media*)

> Bases: `object`

Convenience class over media like photos or documents, which supports accessing the attributes in a more convenient way.

If any of the attributes are not present in the current media, the properties will be `None`.

The original media is available through the `media` attribute.

__dict__ = mappingproxy({'__module__': 'telethon.tl.custom.file', '__doc__': '\n Con

__weakref__

> list of weak references to the object (if defined)

duration

> The duration in seconds of the audio or video.

emoji

> A string with all emoji that represent the current sticker.

ext

> The extension from the mime type of this file.
>
> If the mime type is unknown, the extension from the file name (if any) will be used.

height

> The height in pixels of this media if it's a photo or a video.

id

> The bot-API style `file_id` representing this file.

mime_type

> The mime-type of this file.

name

> The file name of this document.

performer

> The performer of the song.

size

> The size in bytes of this file.

sticker_set

> The InputStickerSet to which the sticker file belongs.

title

> The title of the song.

width
    The width in pixels of this media if it's a photo or a video.

2.35.8 Forward

class telethon.tl.custom.forward.**Forward**(*client*, *original*, *entities*)
    Bases: *telethon.tl.custom.chatgetter.ChatGetter*,   *telethon.tl.custom.sendergetter.SenderGetter*

Custom class that encapsulates a MessageFwdHeader providing an abstraction to easily access information like the original sender.

Remember that this class implements *ChatGetter* and *SenderGetter* which means you have access to all their sender and chat properties and methods.

Attributes:

    original_fwd (MessageFwdHeader): The original MessageFwdHeader instance.

    Any other attribute: Attributes not described here are the same as those available in the original MessageFwdHeader.

2.35.9 InlineBuilder

class telethon.tl.custom.inlinebuilder.**InlineBuilder**(*client*)
    Bases: object

Helper class to allow defining *InlineQuery* results.

Common arguments to all methods are explained here to avoid repetition:

    text (str, optional): If present, the user will send a text message with this text upon being clicked.

    link_preview (bool, optional): Whether to show a link preview in the sent text message or not.

    geo (InputGeoPoint, GeoPoint, InputMediaVenue, MessageMediaVenue, optional): If present, it may either be a geo point or a venue.

    period (int, optional): The period in seconds to be used for geo points.

    contact (InputMediaContact, MessageMediaContact, optional): If present, it must be the contact information to send.

    game (bool, optional): May be True to indicate that the game will be sent.

    buttons (list, custom.Button, KeyboardButton, optional): Same   as   buttons   for *client.send_message()*.

    parse_mode (str, optional): Same as parse_mode for *client.send_message()*.

    id (str, optional): The string ID to use for this result. If not present, it will be the SHA256 hexadecimal digest of converting the created InputBotInlineResult with empty ID to bytes(), so that the ID will be deterministic for the same input.

    Note: If two inputs are exactly the same, their IDs will be the same too. If you send two articles with the same ID, it will raise ResultIdDuplicateError. Consider giving them an explicit ID if you need to send two results that are the same.

    __dict__ = mappingproxy({'__module__': 'telethon.tl.custom.inlinebuilder', '__doc__':

> **__weakref__**
>> list of weak references to the object (if defined)

> **article**(*title*, *description=None*, ***, *url=None*, *thumb=None*, *content=None*, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
>> Creates new inline result of article type.
>>
>> **Args:**
>>
>>> **title (str):** The title to be shown for this result.
>>>
>>> **description (str, optional):** Further explanation of what this result means.
>>>
>>> **url (str, optional):** The URL to be shown for this result.
>>>
>>> **thumb (InputWebDocument, optional):** The thumbnail to be shown for this result. For now it has to be a InputWebDocument if present.
>>>
>>> **content (InputWebDocument, optional):** The content to be shown for this result. For now it has to be a InputWebDocument if present.

> **document**(*file*, *title=None*, ***, *description=None*, *type=None*, *mime_type=None*, *attributes=None*, *force_document=False*, *voice_note=False*, *video_note=False*, *use_cache=True*, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
>> Creates a new inline result of document type.
>>
>> `use_cache`, `mime_type`, `attributes`, `force_document`, `voice_note` and `video_note` are described in `client.send_file`.
>>
>> **Args:**
>>
>>> **file (obj):** Same as `file` for `client.send_file()`.
>>>
>>> **title (str, optional):** The title to be shown for this result.
>>>
>>> **description (str, optional):** Further explanation of what this result means.
>>>
>>> **type (str, optional):** The type of the document. May be one of: photo, gif, mpeg4_gif, video, audio, voice, document, sticker.
>>>
>>> See "Type of the result" in https://core.telegram.org/bots/api.

> **game**(*short_name*, ***, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
>> Creates a new inline result of game type.
>>
>> **Args:**
>>
>>> **short_name (str):** The short name of the game to use.

> **photo**(*file*, ***, *id=None*, *text=None*, *parse_mode=()*, *link_preview=True*, *geo=None*, *period=60*, *contact=None*, *game=False*, *buttons=None*)
>> Creates a new inline result of photo type.
>>
>> **Args:**
>>
>>> **file (obj, optional):** Same as `file` for `client.send_file()`.

2.35.10 InlineResult

class telethon.tl.custom.inlineresult.**InlineResult**(*client*, *original*, *query_id=None*)
> Bases: object

Custom class that encapsulates a bot inline result providing an abstraction to easily access some commonly needed features (such as clicking a result to select it).

Attributes:

>   **result ([BotInlineResult](#)):** The original [BotInlineResult](#) object.

ARTICLE = 'article'

AUDIO = 'audio'

CONTACT = 'contact'

DOCUMENT = 'document'

GAME = 'game'

GIF = 'gif'

LOCATION = 'location'

PHOTO = 'photo'

VENUE = 'venue'

VIDEO = 'video'

VIDEO_GIF = 'mpeg4_gif'

__dict__ = mappingproxy({'__module__': 'telethon.tl.custom.inlineresult', '__doc__':

__weakref__
>   list of weak references to the object (if defined)

click(*entity*, *reply_to=None*, *silent=False*, *clear_draft=False*, *hide_via=False*)
>   Clicks this result and sends the associated [*message*](#).

>   **Args:**

>>   **entity (`entity`):** The entity to which the message of this result should be sent.

>>   **reply_to (`int` | *`Message`*, optional):** If present, the sent message will reply to this ID or message.

>>   **silent (`bool`, optional):** If `True`, the sent message will not notify the user(s).

>>   **clear_draft (`bool`, optional):** Whether the draft should be removed after sending the message from this result or not. Defaults to `False`.

>>   **hide_via (`bool`, optional):** Whether the "via @bot" should be hidden or not. Only works with certain bots (like @bing or @gif).

description
>   The description for this inline result. It may be `None`.

document
>   Returns either the [WebDocument](#) content for normal results or the [Document](#) for media results.

download_media(**args*, ***kwargs*)
>   Downloads the media in this result (if there is a document, the document will be downloaded; otherwise, the photo will if present).

>   This is a wrapper around [`client.download_media`](#).

message
>   The always-present [BotInlineMessage](#) that will be sent if [`click`](#) is called on this result.

photo
>   Returns either the [WebDocument](#) thumbnail for normal results or the [Photo](#) for media results.

title
> The title for this inline result. It may be `None`.

type
> The always-present type of this result. It will be one of: `'article'`, `'photo'`, `'gif'`, `'mpeg4_gif'`, `'video'`, `'audio'`, `'voice'`, `'document'`, `'location'`, `'venue'`, `'contact'`, `'game'`.
>
> You can access all of these constants through *InlineResult*, such as *InlineResult.ARTICLE*, *InlineResult.VIDEO_GIF*, etc.

url
> The URL present in this inline results. If you want to "click" this URL to open it in your browser, you should use Python's `webbrowser.open(url)` for such task.

2.35.11 InlineResults

class telethon.tl.custom.inlineresults.**InlineResults**(*client*, *original*)
> Bases: `list`
>
> Custom class that encapsulates BotResults providing an abstraction to easily access some commonly needed features (such as clicking one of the results to select it)
>
> Note that this is a list of *InlineResult* so you can iterate over it or use indices to access its elements. In addition, it has some attributes.
>
> **Attributes:**
>
> > **result (BotResults):** The original BotResults object.
> >
> > **query_id (`int`):** The random ID that identifies this query.
> >
> > **cache_time (`int`):** For how long the results should be considered valid. You can call *results_valid* at any moment to determine if the results are still valid or not.
> >
> > **users (User):** The users present in this inline query.
> >
> > **gallery (`bool`):** Whether these results should be presented in a grid (as a gallery of images) or not.
> >
> > **next_offset (`str`, optional):** The string to be used as an offset to get the next chunk of results, if any.
> >
> > **switch_pm (InlineBotSwitchPM, optional):** If presents, the results should show a button to switch to a private conversation with the bot using the text in this object.
>
> **__dict__** = mappingproxy({'__module__': 'telethon.tl.custom.inlineresults', '__doc__':
>
> **__repr__**()
> > Return repr(self).
>
> **__str__**()
> > Return str(self).
>
> **__weakref__**
> > list of weak references to the object (if defined)
>
> **results_valid**()
> > Returns `True` if the cache time has not expired yet and the results can still be considered valid.

2.35.12 Message

class telethon.tl.custom.message.**Message**(*id*, *to_id=None*, *date=None*, *out=None*, *mentioned=None*, *media_unread=None*, *silent=None*, *post=None*, *from_id=None*, *reply_to_msg_id=None*, *message=None*, *fwd_from=None*, *via_bot_id=None*, *media=None*, *reply_markup=None*, *entities=None*, *views=None*, *edit_date=None*, *post_author=None*, *grouped_id=None*, *from_scheduled=None*, *legacy=None*, *action=None*)

Bases: *telethon.tl.custom.chatgetter.ChatGetter*, *telethon.tl.custom.sendergetter.SenderGetter*, telethon.tl.tlobject.TLObject, abc.ABC

This custom class aggregates both Message and MessageService to ease accessing their members.

Remember that this class implements *ChatGetter* and *SenderGetter* which means you have access to all their sender and chat properties and methods.

Members:

> **id (`int`):** The ID of this message. This field is *always* present. Any other member is optional and may be `None`.

> **out (`bool`):** Whether the message is outgoing (i.e. you sent it from another session) or incoming (i.e. someone else sent it).
>
> > Note that messages in your own chat are always incoming, but this member will be `True` if you send a message to your own chat. Messages you forward to your chat are *not* considered outgoing, just like official clients display them.

> **mentioned (`bool`):** Whether you were mentioned in this message or not. Note that replies to your own messages also count as mentions.

> **media_unread (`bool`):** Whether you have read the media in this message or not, e.g. listened to the voice note media.

> **silent (`bool`):** Whether this message should notify or not, used in channels.

> **post (`bool`):** Whether this message is a post in a broadcast channel or not.

> **from_scheduled (`bool`):** Whether this message was originated from a scheduled one or not.

> **legacy (`bool`):** Whether this is a legacy message or not.

> **to_id (Peer):** The peer to which this message was sent, which is either PeerUser, PeerChat or PeerChannel. This will always be present except for empty messages.

> **date (`datetime`):** The UTC+0 `datetime` object indicating when this message was sent. This will always be present except for empty messages.

> **message (`str`):** The string text of the message for *Message* instances, which will be `None` for other types of messages.

> **action (MessageAction):** The message action object of the message for MessageService instances, which will be `None` for other types of messages.

> **from_id (`int`):** The ID of the user who sent this message. This will be `None` if the message was sent in a broadcast channel.

> **reply_to_msg_id (`int`):** The ID to which this message is replying to, if any.

fwd_from (**MessageFwdHeader**): The original forward header if this message is a forward. You should probably use the *forward* property instead.

via_bot_id (**int**): The ID of the bot used to send this message through its inline mode (e.g. "via @like").

media (**MessageMedia**): The media sent with this message if any (such as photos, videos, documents, gifs, stickers, etc.).

You may want to access the *photo*, *document* etc. properties instead.

If the media was not present or it was MessageMediaEmpty, this member will instead be None for convenience.

reply_markup (**ReplyMarkup**): The reply markup for this message (which was sent either via a bot or by a bot). You probably want to access *buttons* instead.

entities (**List[MessageEntity]**): The list of markup entities in this message, such as bold, italics, code, hyperlinks, etc.

views (**int**): The number of views this message from a broadcast channel has. This is also present in forwards.

edit_date (**datetime**): The date when this message was last edited.

post_author (**str**): The display name of the message sender to show in messages sent to broadcast channels.

grouped_id (**int**): If this message belongs to a group of messages (photo albums or video albums), all of them will have the same value here.

action_entities
Returns a list of entities that took part in this action.

Possible cases for this are MessageActionChatAddUser, types.MessageActionChatCreate, MessageActionChatDeleteUser, MessageActionChatJoinedByLink MessageActionChatMigrateTo and MessageActionChannelMigrateFrom.

If the action is neither of those, the result will be None. If some entities could not be retrieved, the list may contain some None items in it.

audio
The Document media in this message, if it's an audio file.

button_count
Returns the total button count (sum of all *buttons* rows).

buttons
Returns a list of lists of *MessageButton*, if any.

Otherwise, it returns None.

click (*i=None, j=None, *, text=None, filter=None, data=None*)
Calls *button.click* on the specified button.

Does nothing if the message has no buttons.

Args:

i (**int**): Clicks the i'th button (starting from the index 0). Will raise IndexError if out of bounds. Example:

```
>>> message = ...  # get the message somehow
>>> # Clicking the 3rd button
>>> # [button1] [button2]
```

(continues on next page)

```
>>> # [      button3      ]
>>> # [button4] [button5]
>>> message.click(2)   # index
```

j (**int**): Clicks the button at position (i, j), these being the indices for the (row, column) respectively. Example:

```
>>> # Clicking the 2nd button on the 1st row.
>>> # [button1] [button2]
>>> # [      button3      ]
>>> # [button4] [button5]
>>> message.click(0, 1)   # (row, column)
```

This is equivalent to `message.buttons[0][1].click()`.

text (**str** | **callable**): Clicks the first button with the text "text". This may also be a callable, like a `re.compile(...).match`, and the text will be passed to it.

filter (**callable**): Clicks the first button for which the callable returns `True`. The callable should accept a single *MessageButton* argument.

data (**bytes**): This argument overrides the rest and will not search any buttons. Instead, it will directly send the request to behave as if it clicked a button with said data. Note that if the message does not have this data, it will `raise DataInvalidError`.

Example:

```
# Click the first button
message.click(0)

# Click some row/column
message.click(row, column)

# Click by text
message.click(text='')

# Click by data
message.click(data=b'payload')
```

client

Returns the *TelegramClient* that *patched* this message. This will only be present if you **use the friendly methods**, it won't be there if you invoke raw API methods manually, in which case you should only access members, not properties.

contact

The [MessageMediaContact](#) in this message, if it's a contact.

delete(**args*, ***kwargs*)

Deletes the message. You're responsible for checking whether you have the permission to do so, or to except the error otherwise. Shorthand for `telethon.client.messages.MessageMethods.delete_messages` with `entity` and `message_ids` already set.

If you need to delete more than one message at once, don't use this *delete* method. Use a *telethon. client.telegramclient.TelegramClient* instance directly.

document

The [Document](#) media in this message, if any.

---

download_media(**args*, ***kwargs*)

Downloads the media contained in the message, if any. Shorthand for *telethon.client.
downloads.DownloadMethods.download_media* with the `message` already set.

edit(**args*, ***kwargs*)

Edits the message iff it's outgoing. Shorthand for *telethon.client.messages.
MessageMethods.edit_message* with both `entity` and `message` already set.

Returns *None* if the message was incoming, or the edited *Message* otherwise.

Note: This is different from *client.edit_message* and **will respect** the previous state of the message. For example, if the message didn't have a link preview, the edit won't add one by default, and you should force it by setting it to *True* if you want it.

This is generally the most desired and convenient behaviour, and will work for link previews and message buttons.

---

file

Returns a *File* wrapping the *photo* or *document* in this message. If the media type is different (polls, games, none, etc.), this property will be *None*.

This instance lets you easily access other properties, such as *file.id*, *file.name*, etc., without having to manually inspect the `document.attributes`.

forward

The *Forward* information if this message is a forwarded message.

forward_to(**args*, ***kwargs*)

Forwards the message. Shorthand for *telethon.client.messages.MessageMethods.
forward_messages* with both `messages` and `from_peer` already set.

If you need to forward more than one message at once, don't use this *forward_to* method. Use a *telethon.client.telegramclient.TelegramClient* instance directly.

game

The Game media in this message, if it's a game.

geo

The GeoPoint media in this message, if it has a location.

get_buttons()

Returns *buttons* when that property fails (this is rarely needed).

get_entities_text(*cls=None*)

Returns a list of (`markup entity`, `inner text`) (like bold or italics).

The markup entity is a MessageEntity that represents bold, italics, etc., and the inner text is the `str` inside that markup entity.

For example:

```
print(repr(message.text))  # shows: 'Hello **world**!'


for ent, txt in message.get_entities_text():
    print(ent)  # shows: MessageEntityBold(offset=6, length=5)
    print(txt)  # shows: world
```

Args:

---

cls (`type`): Returns entities matching this type only. For example, the following will print the text for all `code` entities:

```
>>> from telethon.tl.types import MessageEntityCode
>>>
>>> m = ...  # get the message
>>> for _, inner_text in m.get_entities_text(MessageEntityCode):
>>>     print(inner_text)
```

get_reply_message()
> The *Message* that this message is replying to, or *None*.
>
> The result will be cached after its first use.

gif
> The Document media in this message, if it's a "gif".
>
> "Gif" files by Telegram are normally `.mp4` video files without sound, the so called "animated" media. However, it may be the actual gif format if the file is too large.

invoice
> The MessageMediaInvoice in this message, if it's an invoice.

is_reply
> *True* if the message is a reply to some other message.
>
> Remember that you can access the ID of the message this one is replying to through `reply_to_msg_id`, and the *Message* object with *get_reply_message()*.

mark_read()
> Marks the message as read. Shorthand for *client.send_read_acknowledge()* with both `entity` and `message` already set.

photo
> The Photo media in this message, if any.
>
> This will also return the photo for MessageService if its action is MessageActionChatEditPhoto, or if the message has a web preview with a photo.

pin(**, notify=False*)
> Pins the message. Shorthand for *telethon.client.messages.MessageMethods.pin_message* with both `entity` and `message` already set.

poll
> The MessageMediaPoll in this message, if it's a poll.

raw_text
> The raw message text, ignoring any formatting. Will be *None* for MessageService.
>
> Setting a value to this field will erase the `entities`, unlike changing the `message` member.

reply(**args, **kwargs*)
> Replies to the message (as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with both `entity` and `reply_to` already set.

respond(**args, **kwargs*)
> Responds to the message (not as a reply). Shorthand for *telethon.client.messages.MessageMethods.send_message* with `entity` already set.

sticker
> The Document media in this message, if it's a sticker.

---

text
    The message text, formatted using the client's default parse mode. Will be `None` for MessageService.

venue
    The MessageMediaVenue in this message, if it's a venue.

via_bot
    The bot User if the message was sent via said bot.

    This will only be present if `via_bot_id` is not `None` and the entity is known.

via_input_bot
    Returns the input variant of *via_bot*.

video
    The Document media in this message, if it's a video.

video_note
    The Document media in this message, if it's a video note.

voice
    The Document media in this message, if it's a voice note.

web_preview
    The WebPage media in this message, if any.

2.35.13 MessageButton

class telethon.tl.custom.messagebutton.**MessageButton**(*client*, *original*, *chat*, *bot*, *msg_id*)

    Bases: `object`

Note: *Message.buttons* are instances of this type. If you want to **define** a reply markup for e.g. sending messages, refer to *Button* instead.

Custom class that encapsulates a message button providing an abstraction to easily access some commonly needed features (such as clicking the button itself).

Attributes:

    button (KeyboardButton): The original KeyboardButton object.

__dict__ = mappingproxy({'__module__': 'telethon.tl.custom.messagebutton', '__doc__':

__weakref__
    list of weak references to the object (if defined)

click()
    Emulates the behaviour of clicking this button.

    If it's a normal KeyboardButton with text, a message will be sent, and the sent *Message* returned.

    If it's an inline KeyboardButtonCallback with text and data, it will be "clicked" and the BotCallbackAnswer returned.

    If it's an inline KeyboardButtonSwitchInline button, the StartBotRequest will be invoked and the resulting updates returned.

    If it's a KeyboardButtonUrl, the URL of the button will be passed to `webbrowser.open` and return `True` on success.

client
> Returns the *telethon.client.telegramclient.TelegramClient* instance that created this instance.

data
> The `bytes` data for KeyboardButtonCallback objects.

inline_query
> The query `str` for KeyboardButtonSwitchInline objects.

text
> The text string of the button.

url
> The url `str` for KeyboardButtonUrl objects.

2.35.14 SenderGetter

class telethon.tl.custom.sendergetter.**SenderGetter**(*sender_id=None*, *, *sender=None*, *input_sender=None*)

Bases: `abc.ABC`

Helper base class that introduces the *sender*, *input_sender* and *sender_id* properties and *get_sender* and *get_input_sender* methods.

__dict__ = mappingproxy({'__module__': 'telethon.tl.custom.sendergetter', '__doc__':

__weakref__
> list of weak references to the object (if defined)

get_input_sender()
> Returns *input_sender*, but will make an API call to find the input sender unless it's already cached.

get_sender()
> Returns *sender*, but will make an API call to find the sender unless it's already cached.
>
> If you only need the ID, use *sender_id* instead.
>
> If you need to call a method which needs this sender, use *get_input_sender()* instead.

input_sender
> This InputPeer is the input version of the user/channel who sent the message. Similarly to *input_chat*, this doesn't have things like username or similar, but still useful in some cases.
>
> Note that this might not be available if the library can't find the input chat, or if the message a broadcast on a channel.

sender
> Returns the User or Channel that sent this object. It may be `None` if Telegram didn't send the sender.
>
> If you only need the ID, use *sender_id* instead.
>
> If you need to call a method which needs this chat, use *input_sender* instead.
>
> If you're using *telethon.events*, use *get_sender()* instead.

sender_id
> Returns the marked sender integer ID, if present.
>
> If there is a sender in the object, *sender_id* will *always* be set, which is why you should use it instead of *sender.id*.

2.36 Utilities

These are the utilities that the library has to offer.

Utilities for working with the Telegram API itself (such as handy methods to convert between an entity like a User, Chat, etc. into its Input version)

class `telethon.utils.`**`AsyncClassWrapper`**(*wrapped*)

   Bases: `object`

   `__dict__` `= mappingproxy({'__module__':` `'telethon.utils', '__init__':` `<function Async`

   `__getattr__`(*item*)

   `__weakref__`
      list of weak references to the object (if defined)

`telethon.utils.`**`chunks`**(*iterable*, *size=100*)
   Turns the given iterable into chunks of the specified size, which is 100 by default since that's what Telegram uses the most.

`telethon.utils.`**`decode_waveform`**(*waveform*)
   Inverse operation of *encode_waveform*.

`telethon.utils.`**`encode_waveform`**(*waveform*)
   Encodes the input `bytes` into a 5-bit byte-string to be used as a voice note's waveform. See *decode_waveform* for the reverse operation.

   Example

```
chat = ...
file = 'my.ogg'

# Send 'my.ogg' with a ascending-triangle waveform
client.send_file(chat, file, attributes=[types.DocumentAttributeAudio(
    duration=7,
    voice=True,
    waveform=utils.encode_waveform(bytes(range(2 ** 5))  # 2**5 because 5-bit
)]

# Send 'my.ogg' with a square waveform
client.send_file(chat, file, attributes=[types.DocumentAttributeAudio(
    duration=7,
    voice=True,
    waveform=utils.encode_waveform(bytes((31, 31, 15, 15, 15, 15, 31, 31)) *␣
↪4)
)]
```

`telethon.utils.`**`get_appropriated_part_size`**(*file_size*)
   Gets the appropriated part size when uploading or downloading files, given an initial file size.

`telethon.utils.`**`get_attributes`**(*file*, ***, *attributes=None*, *mime_type=None*, *force_document=False*, *voice_note=False*, *video_note=False*, *supports_streaming=False*)
   Get a list of attributes for the given file and the mime type as a tuple ([attribute], mime_type).

`telethon.utils.`**`get_display_name`**(*entity*)
   Gets the display name for the given User, Chat or Channel. Returns an empty string otherwise.

`telethon.utils.`**`get_extension`**(*media*)
   Gets the corresponding extension for any Telegram media.

telethon.utils.**get_inner_text**(*text*, *entities*)

Gets the inner text that's surrounded by the given entities. For instance: text = 'hey!', entity = MessageEntity-Bold(2, 2) -> 'y!'.

> **Parameters**
>
> > • **text** – the original text.
> >
> > • **entities** – the entity or entities that must be matched.
>
> **Returns** a single result or a list of the text surrounded by the entities.

telethon.utils.**get_input_channel**(*entity*)

Similar to *get_input_peer()*, but for InputChannel's alone.

telethon.utils.**get_input_chat_photo**(*photo*)

Similar to *get_input_peer()*, but for chat photos

telethon.utils.**get_input_dialog**(*dialog*)

Similar to *get_input_peer()*, but for dialogs

telethon.utils.**get_input_document**(*document*)

Similar to *get_input_peer()*, but for documents

telethon.utils.**get_input_geo**(*geo*)

Similar to *get_input_peer()*, but for geo points

telethon.utils.**get_input_location**(*location*)

Similar to *get_input_peer()*, but for input messages.

Note that this returns a tuple (dc_id, location), the dc_id being present if known.

telethon.utils.**get_input_media**(*media*, ***, *is_photo=False*, *attributes=None*, *force_document=False*, *voice_note=False*, *video_note=False*, *supports_streaming=False*)

Similar to *get_input_peer()*, but for media.

If the media is InputFile and is_photo is known to be True, it will be treated as an InputMediaUploaded-Photo. Else, the rest of parameters will indicate how to treat it.

telethon.utils.**get_input_message**(*message*)

Similar to *get_input_peer()*, but for input messages.

telethon.utils.**get_input_peer**(*entity*, *allow_self=True*, *check_hash=True*)

Gets the input peer for the given "entity" (user, chat or channel).

A TypeError is raised if the given entity isn't a supported type or if check_hash is True but the entity's access_hash is None.

Note that check_hash **is ignored** if an input peer is already passed since in that case we assume the user knows what they're doing. This is key to getting entities by explicitly passing hash = 0.

telethon.utils.**get_input_photo**(*photo*)

Similar to *get_input_peer()*, but for photos

telethon.utils.**get_input_user**(*entity*)

Similar to *get_input_peer()*, but for InputUser's alone.

telethon.utils.**get_message_id**(*message*)

Similar to *get_input_peer()*, but for message IDs.

telethon.utils.**get_peer**(*peer*)

telethon.utils.**get_peer_id**(*peer*, *add_mark=True*)

Convert the given peer into its marked ID by default.

This "mark" comes from the "bot api" format, and with it the peer type can be identified back. User ID is left unmodified, chat ID is negated, and channel ID is prefixed with -100:

- `user_id`

- `-chat_id`

- `-100channel_id`

The original ID and the peer type class can be returned with a call to `resolve_id(marked_id)()`.

telethon.utils.**is_audio**(*file*)
  Returns `True` if the file extension looks like an audio file.

telethon.utils.**is_gif**(*file*)
  Returns `True` if the file extension looks like a gif file to Telegram.

telethon.utils.**is_image**(*file*)
  Returns `True` if the file extension looks like an image file to Telegram.

telethon.utils.**is_list_like**(*obj*)
  Returns `True` if the given object looks like a list.

  Checking if `hasattr(obj, '__iter__')` and ignoring `str`/`bytes` is not enough. Things like `open()` are also iterable (and probably many other things), so just support the commonly known list-like objects.

telethon.utils.**is_video**(*file*)
  Returns `True` if the file extension looks like a video file.

telethon.utils.**pack_bot_file_id**(*file*)
  Inverse operation for *resolve_bot_file_id*.

  The only parameters this method will accept are Document and Photo, and it will return a variable-length `file_id` string.

  If an invalid parameter is given, it will `return None`.

telethon.utils.**parse_phone**(*phone*)
  Parses the given phone, or returns `None` if it's invalid.

telethon.utils.**parse_username**(*username*)
  Parses the given username or channel access hash, given a string, username or URL. Returns a tuple consisting of both the stripped, lowercase username and whether it is a joinchat/ hash (in which case is not lowercase'd).

  Returns `(None, False)` if the `username` or link is not valid.

telethon.utils.**resolve_bot_file_id**(*file_id*)
  Given a Bot API-style *file_id*, returns the media it represents. If the *file_id* is not valid, `None` is returned instead.

  Note that the *file_id* does not have information such as image dimensions or file size, so these will be zero if present.

  For thumbnails, the photo ID and hash will always be zero.

telethon.utils.**resolve_id**(*marked_id*)
  Given a marked ID, returns the original ID and its Peer type.

telethon.utils.**resolve_inline_message_id**(*inline_msg_id*)
  Resolves an inline message ID. Returns a tuple of (`message id, peer, dc id, access hash`)

  The `peer` may either be a PeerUser referencing the user who sent the message via the bot in a private conversation or small group chat, or a PeerChannel if the message was sent in a channel.

The `access_hash` does not have any use yet.

telethon.utils.**resolve_invite_link**(*link*)
> Resolves the given invite link. Returns a tuple of (`link creator user id, global chat id, random int`).

> Note that for broadcast channels, the link creator user ID will be zero to protect their identity. Normal chats and megagroup channels will have such ID.

> Note that the chat ID may not be accurate for chats with a link that were upgraded to megagroup, since the link can remain the same, but the chat ID will be correct once a new link is generated.

telethon.utils.**sanitize_parse_mode**(*mode*)
> Converts the given parse mode into an object with `parse` and `unparse` callable properties.

telethon.utils.**stripped_photo_to_jpg**(*stripped*)
> Adds the JPG header and footer to a stripped image.

> Ported from https://github.com/telegramdesktop/tdesktop/blob/bec39d89e19670eb436dc794a8f20b657cb87c71/Telegram/SourceFiles/ui/image/image.cpp#L225

2.37 API Errors

These are the base errors that Telegram's API may raise.

See *RPC Errors* for a more friendly explanation.

Errors not related to the Telegram API itself

exception telethon.errors.common.**AlreadyInConversationError**
> Bases: `Exception`

> Occurs when another exclusive conversation is opened in the same chat.

> **__weakref__**
> > list of weak references to the object (if defined)

exception telethon.errors.common.**BadMessageError**(*request*, *code*)
> Bases: `Exception`

> Occurs when handling a bad_message_notification.

> **ErrorMessages = {16: 'msg_id too low (most likely, client time is wrong it would be w**

> **__weakref__**
> > list of weak references to the object (if defined)

exception telethon.errors.common.**CdnFileTamperedError**
> Bases: *telethon.errors.common.SecurityError*

> Occurs when there's a hash mismatch between the decrypted CDN file and its expected hash.

exception telethon.errors.common.**InvalidBufferError**(*payload*)
> Bases: `BufferError`

> Occurs when the buffer is invalid, and may contain an HTTP error code. For instance, 404 means "forgotten/broken authorization key", while

> **__weakref__**
> > list of weak references to the object (if defined)

exception `telethon.errors.common.`**InvalidChecksumError**(*checksum*, *valid_checksum*)
> Bases: `Exception`

> Occurs when using the TCP full mode and the checksum of a received packet doesn't match the expected checksum.

> **__weakref__**
>> list of weak references to the object (if defined)

exception `telethon.errors.common.`**MultiError**
> Bases: `Exception`

> Exception container for multiple `TLRequest`'s.

> **static __new__**(*cls*, *exceptions*, *result*, *requests*)
>> Create and return a new object. See help(type) for accurate signature.

> **__weakref__**
>> list of weak references to the object (if defined)

exception `telethon.errors.common.`**ReadCancelledError**
> Bases: `Exception`

> Occurs when a read operation was cancelled.

> **__weakref__**
>> list of weak references to the object (if defined)

exception `telethon.errors.common.`**SecurityError**(**args*)
> Bases: `Exception`

> Generic security error, mostly used when generating a new AuthKey.

> **__weakref__**
>> list of weak references to the object (if defined)

exception `telethon.errors.common.`**TypeNotFoundError**(*invalid_constructor_id*, *remaining*)
> Bases: `Exception`

> Occurs when a type is not found, for example, when trying to read a TLObject with an invalid constructor code.

> **__weakref__**
>> list of weak references to the object (if defined)

exception `telethon.errors.rpcbaseerrors.`**AuthKeyError**(*request*, *message*, *code=None*)
> Bases: `telethon.errors.rpcbaseerrors.RPCError`

> Errors related to invalid authorization key, like AUTH_KEY_DUPLICATED which can cause the connection to fail.

> **code = 406**

> **message = 'AUTH_KEY'**

exception `telethon.errors.rpcbaseerrors.`**BadRequestError**(*request*, *message*, *code=None*)
> Bases: `telethon.errors.rpcbaseerrors.RPCError`

> The query contains errors. In the event that a request was created using a form and contains user generated data, the user should be notified that the data must be corrected before the query is repeated.

> **code = 400**

> **message = 'BAD_REQUEST'**

`telethon.errors.rpcbaseerrors.`**`BotTimeout`**
    alias of *`telethon.errors.rpcbaseerrors.TimedOutError`*

exception `telethon.errors.rpcbaseerrors.`**`FloodError`**(*request*, *message*, *code=None*)
    Bases: *`telethon.errors.rpcbaseerrors.RPCError`*

    The maximum allowed number of attempts to invoke the given method with the given input parameters has been exceeded. For example, in an attempt to request a large number of text messages (SMS) for the same phone number.

    `code = 420`

    `message = 'FLOOD'`

exception `telethon.errors.rpcbaseerrors.`**`ForbiddenError`**(*request*,   *message*, *code=None*)
    Bases: *`telethon.errors.rpcbaseerrors.RPCError`*

    Privacy violation. For example, an attempt to write a message to someone who has blacklisted the current user.

    `code = 403`

    `message = 'FORBIDDEN'`

exception `telethon.errors.rpcbaseerrors.`**`InvalidDCError`**(*request*,   *message*, *code=None*)
    Bases: *`telethon.errors.rpcbaseerrors.RPCError`*

    The request must be repeated, but directed to a different data center.

    `code = 303`

    `message = 'ERROR_SEE_OTHER'`

exception `telethon.errors.rpcbaseerrors.`**`NotFoundError`**(*request*,   *message*, *code=None*)
    Bases: *`telethon.errors.rpcbaseerrors.RPCError`*

    An attempt to invoke a non-existent object, such as a method.

    `code = 404`

    `message = 'NOT_FOUND'`

exception `telethon.errors.rpcbaseerrors.`**`RPCError`**(*request*, *message*, *code=None*)
    Bases: `Exception`

    Base class for all Remote Procedure Call errors.

    `__reduce__`()
        Helper for pickle.

    `__weakref__`
        list of weak references to the object (if defined)

    `code = None`

    `message = None`

exception `telethon.errors.rpcbaseerrors.`**`ServerError`**(*request*, *message*, *code=None*)
    Bases: *`telethon.errors.rpcbaseerrors.RPCError`*

    An internal server error occurred while a request was being processed for example, there was a disruption while accessing a database or file storage.

    `code = 500`

    `message = 'INTERNAL'`

exception telethon.errors.rpcbaseerrors.**TimedOutError**(*request*, *message*, *code=None*)

Bases: *telethon.errors.rpcbaseerrors.RPCError*

Clicking the inline buttons of bots that never (or take to long to) call `answerCallbackQuery` will result in this "special" RPCError.

code = 503

message = 'Timeout'

exception telethon.errors.rpcbaseerrors.**UnauthorizedError**(*request*, *message*, *code=None*)

Bases: *telethon.errors.rpcbaseerrors.RPCError*

There was an unauthorized attempt to use functionality available only to authorized users.

code = 401

message = 'UNAUTHORIZED'

2.38 Sessions

These are the different built-in session storage that you may subclass.

class telethon.sessions.abstract.**Session**

Bases: abc.ABC

__dict__ = mappingproxy({'__module__': 'telethon.sessions.abstract', '__init__': <fu

__weakref__
list of weak references to the object (if defined)

auth_key
Returns an `AuthKey` instance associated with the saved data center, or `None` if a new one should be generated.

cache_file(*md5_digest*, *file_size*, *instance*)
Caches the given file information persistently, so that it doesn't need to be re-uploaded in case the file is used again.

The `instance` will be either an `InputPhoto` or `InputDocument`, both with an `.id` and `.access_hash` attributes.

clone(*to_instance=None*)
Creates a clone of this session file.

close()
Called on client disconnection. Should be used to free any used resources. Can be left empty if none.

dc_id
Returns the currently-used data center ID.

delete()
Called upon client.log_out(). Should delete the stored information from disk since it's not valid anymore.

get_file(*md5_digest*, *file_size*, *cls*)
Returns an instance of `cls` if the `md5_digest` and `file_size` match an existing saved record. The class will either be an `InputPhoto` or `InputDocument`, both with two parameters `id` and `access_hash` in that order.

get_input_entity(*key*)
> Turns the given key into an `InputPeer` (e.g. `InputPeerUser`). The library uses this method whenever an `InputPeer` is needed to suit several purposes (e.g. user only provided its ID or wishes to use a cached username to avoid extra RPC).

get_update_state(*entity_id*)
> Returns the `UpdateState` associated with the given `entity_id`. If the `entity_id` is 0, it should return the `UpdateState` for no specific channel (the "general" state). If no state is known it should `return None`.

classmethod list_sessions()
> Lists available sessions. Not used by the library itself.

port
> Returns the port to which the library should connect to.

process_entities(*tlo*)
> Processes the input `TLObject` or `list` and saves whatever information is relevant (e.g., ID or access hash).

save()
> Called whenever important properties change. It should make persist the relevant session information to disk.

server_address
> Returns the server address where the library should connect to.

set_dc(*dc_id*, *server_address*, *port*)
> Sets the information of the data center address and port that the library should connect to, as well as the data center ID, which is currently unused.

set_update_state(*entity_id*, *state*)
> Sets the given `UpdateState` for the specified `entity_id`, which should be 0 if the `UpdateState` is the "general" state (and not for any specific channel).

takeout_id
> Returns an ID of the takeout process initialized for this session, or `None` if there's no were any unfinished takeout requests.

class telethon.sessions.memory.**MemorySession**
> Bases: *telethon.sessions.abstract.Session*

auth_key
> Returns an `AuthKey` instance associated with the saved data center, or `None` if a new one should be generated.

cache_file(*md5_digest*, *file_size*, *instance*)
> Caches the given file information persistently, so that it doesn't need to be re-uploaded in case the file is used again.
>
> The `instance` will be either an `InputPhoto` or `InputDocument`, both with an `.id` and `.access_hash` attributes.

close()
> Called on client disconnection. Should be used to free any used resources. Can be left empty if none.

dc_id
> Returns the currently-used data center ID.

delete()
> Called upon client.log_out(). Should delete the stored information from disk since it's not valid anymore.

---

get_entity_rows_by_id(*id*, *exact=True*)

get_entity_rows_by_name(*name*)

get_entity_rows_by_phone(*phone*)

get_entity_rows_by_username(*username*)

get_file(*md5_digest*, *file_size*, *cls*)
>   Returns an instance of `cls` if the `md5_digest` and `file_size` match an existing saved record. The class will either be an `InputPhoto` or `InputDocument`, both with two parameters `id` and `access_hash` in that order.

get_input_entity(*key*)
>   Turns the given key into an `InputPeer` (e.g. `InputPeerUser`). The library uses this method whenever an `InputPeer` is needed to suit several purposes (e.g. user only provided its ID or wishes to use a cached username to avoid extra RPC).

get_update_state(*entity_id*)
>   Returns the `UpdateState` associated with the given `entity_id`. If the `entity_id` is 0, it should return the `UpdateState` for no specific channel (the "general" state). If no state is known it should `return None`.

port
>   Returns the port to which the library should connect to.

process_entities(*tlo*)
>   Processes the input `TLObject` or `list` and saves whatever information is relevant (e.g., ID or access hash).

save()
>   Called whenever important properties change. It should make persist the relevant session information to disk.

server_address
>   Returns the server address where the library should connect to.

set_dc(*dc_id*, *server_address*, *port*)
>   Sets the information of the data center address and port that the library should connect to, as well as the data center ID, which is currently unused.

set_update_state(*entity_id*, *state*)
>   Sets the given `UpdateState` for the specified `entity_id`, which should be 0 if the `UpdateState` is the "general" state (and not for any specific channel).

takeout_id
>   Returns an ID of the takeout process initialized for this session, or `None` if there's no were any unfinished takeout requests.

class telethon.sessions.sqlite.**SQLiteSession**(*session_id=None*)
>   Bases: *telethon.sessions.memory.MemorySession*

>   This session contains the required information to login into your Telegram account. NEVER give the saved session file to anyone, since they would gain instant access to all your messages and contacts.

>   If you think the session has been compromised, close all the sessions through an official Telegram client to revoke the authorization.

>   **auth_key**

>   **cache_file**(*md5_digest*, *file_size*, *instance*)
>   >   Caches the given file information persistently, so that it doesn't need to be re-uploaded in case the file is used again.

The `instance` will be either an `InputPhoto` or `InputDocument`, both with an `.id` and `.access_hash` attributes.

clone(*to_instance=None*)
Creates a clone of this session file.

close()
Closes the connection unless we're working in-memory

delete()
Deletes the current session file

get_entity_rows_by_id(*id*, *exact=True*)

get_entity_rows_by_name(*name*)

get_entity_rows_by_phone(*phone*)

get_entity_rows_by_username(*username*)

get_file(*md5_digest*, *file_size*, *cls*)
Returns an instance of `cls` if the `md5_digest` and `file_size` match an existing saved record. The class will either be an `InputPhoto` or `InputDocument`, both with two parameters `id` and `access_hash` in that order.

get_update_state(*entity_id*)
Returns the `UpdateState` associated with the given `entity_id`. If the `entity_id` is 0, it should return the `UpdateState` for no specific channel (the "general" state). If no state is known it should `return None`.

classmethod list_sessions()
Lists all the sessions of the users who have ever connected using this client and never logged out

process_entities(*tlo*)
Processes all the found entities on the given TLObject, unless .enabled is False.

Returns True if new input entities were added.

save()
Saves the current session object as session_user_id.session

set_dc(*dc_id*, *server_address*, *port*)
Sets the information of the data center address and port that the library should connect to, as well as the data center ID, which is currently unused.

set_update_state(*entity_id*, *state*)
Sets the given `UpdateState` for the specified `entity_id`, which should be 0 if the `UpdateState` is the "general" state (and not for any specific channel).

takeout_id

class telethon.sessions.string.**StringSession**(*string: str = None*)
Bases: *telethon.sessions.memory.MemorySession*

This session file can be easily saved and loaded as a string. According to the initial design, it contains only the data that is necessary for successful connection and authentication, so takeout ID is not stored.

It is thought to be used where you don't want to create any on-disk files but would still like to be able to save and load existing sessions by other means.

You can use custom *encode* and *decode* functions, if present:

- *encode* definition must be `def encode(value:  bytes) -> str:`.

- *decode* definition must be `def decode(value:  str) -> bytes:`.

---

static decode (*x: str*) → bytes

static encode (*x: bytes*) → str

save ()
> Called whenever important properties change. It should make persist the relevant session information to disk.

2.39 Connection Modes

The only part about network that you should worry about are the different connection modes, which are the following:

class telethon.network.connection.tcpfull.**ConnectionTcpFull** (*ip*, *port*, *dc_id*, ***, *loop*, *loggers*, *proxy=None*)

> Bases: telethon.network.connection.connection.Connection
>
> Default Telegram mode. Sends 12 additional bytes and needs to calculate the CRC value of the packet itself.
>
> **packet_codec**
> > alias of *FullPacketCodec*

class telethon.network.connection.tcpfull.**FullPacketCodec** (*connection*)
> Bases: telethon.network.connection.connection.PacketCodec
>
> **encode_packet** (*data*)
> > Encodes single packet and returns encoded bytes.
>
> **read_packet** (*reader*)
> > Reads single packet from reader object that should have readexactly(n) method.
>
> **tag = None**

class telethon.network.connection.tcpabridged.**AbridgedPacketCodec** (*connection*)
> Bases: telethon.network.connection.connection.PacketCodec
>
> **encode_packet** (*data*)
> > Encodes single packet and returns encoded bytes.
>
> **obfuscate_tag = b'\xef\xef\xef\xef'**
>
> **read_packet** (*reader*)
> > Reads single packet from reader object that should have readexactly(n) method.
>
> **tag = b'\xef'**

class telethon.network.connection.tcpabridged.**ConnectionTcpAbridged** (*ip*, *port*, *dc_id*, ***, *loop*, *loggers*, *proxy=None*)

> Bases: telethon.network.connection.connection.Connection
>
> This is the mode with the lowest overhead, as it will only require 1 byte if the packet length is less than 508 bytes (127 << 2, which is very common).
>
> **packet_codec**
> > alias of *AbridgedPacketCodec*

class telethon.network.connection.tcpintermediate.**ConnectionTcpIntermediate**(*ip,*
port,
dc_id,
*,
loop,
log-
gers,
proxy=None)

    Bases: telethon.network.connection.connection.Connection

    Intermediate mode between ConnectionTcpFull and ConnectionTcpAbridged. Always sends 4 extra bytes for the packet length.

    packet_codec
        alias of *[IntermediatePacketCodec](#)*

class telethon.network.connection.tcpintermediate.**IntermediatePacketCodec**(*connection*)

    Bases: telethon.network.connection.connection.PacketCodec

    encode_packet(*data*)
        Encodes single packet and returns encoded bytes.

    obfuscate_tag = b'\xee\xee\xee\xee'

    read_packet(*reader*)
        Reads single packet from reader object that should have readexactly(n) method.

    tag = b'\xee\xee\xee\xee'

class telethon.network.connection.tcpintermediate.**RandomizedIntermediatePacketCodec**(*connectio*

    Bases: *[telethon.network.connection.tcpintermediate.IntermediatePacketCodec](#)*

    Data packets are aligned to 4bytes. This codec adds random bytes of size from 0 to 3 bytes, which are ignored by decoder.

    encode_packet(*data*)
        Encodes single packet and returns encoded bytes.

    obfuscate_tag = b'\xdd\xdd\xdd\xdd'

    read_packet(*reader*)
        Reads single packet from reader object that should have readexactly(n) method.

    tag = None

class telethon.network.connection.tcpobfuscated.**ConnectionTcpObfuscated**(*ip,*
port,
dc_id,
*,
loop,
log-
gers,
proxy=None)

    Bases: telethon.network.connection.connection.ObfuscatedConnection

    Mode that Telegram defines as "obfuscated2". Encodes the packet just like ConnectionTcpAbridged, but encrypts every message with a randomly generated key using the AES-CTR mode so the packets are harder to discern.

    obfuscated_io
        alias of *[ObfuscatedIO](#)*

> **packet_codec**
>> alias of *telethon.network.connection.tcpabridged.AbridgedPacketCodec*

class telethon.network.connection.tcpobfuscated.**ObfuscatedIO**(*connection*)
> Bases: object

> **__dict__** = mappingproxy({'__module__': 'telethon.network.connection.tcpobfuscated', '

> **__weakref__**
>> list of weak references to the object (if defined)

> **header = None**

> **static init_header**(*packet_codec*)

> **readexactly**(*n*)

> **write**(*data*)

class telethon.network.connection.http.**ConnectionHttp**(*ip*, *port*, *dc_id*, ***, *loop*, *loggers*, *proxy=None*)
> Bases: telethon.network.connection.connection.Connection

> **connect**(*timeout=None*, *ssl=None*)
>> Establishes a connection with the server.

> **packet_codec**
>> alias of *HttpPacketCodec*

class telethon.network.connection.http.**HttpPacketCodec**(*connection*)
> Bases: telethon.network.connection.connection.PacketCodec

> **encode_packet**(*data*)
>> Encodes single packet and returns encoded bytes.

> **obfuscate_tag = None**

> **read_packet**(*reader*)
>> Reads single packet from reader object that should have readexactly(n) method.

> **tag = None**

2.40 Helpers

Various helpers not related to the Telegram API itself

class telethon.helpers.**TotalList**(**args*, ***kwargs*)
> Bases: list

A list with an extra total property, which may not match its len since the total represents the total amount of items *available* somewhere else, not the items *in this list*.

Examples:

```
# Telethon returns these lists in some cases (for example,
# only when a chunk is returned, but the "total" count
# is available).
result = client.get_messages(chat, limit=10)

print(result.total)  # large number
print(len(result))  # 10
```

(continues on next page)

```python
print(result[0])  # latest message

for x in result:  # show the 10 messages
    print(x.text)
```

__dict__ = mappingproxy({'__module__': 'telethon.helpers', '__doc__': '\n A list with

__repr__()
> Return repr(self).

__str__()
> Return str(self).

__weakref__
> list of weak references to the object (if defined)

telethon.helpers.**add_surrogate**(*text*)

telethon.helpers.**del_surrogate**(*text*)

telethon.helpers.**ensure_parent_dir_exists**(*file_path*)
> Ensures that the parent directory exists

telethon.helpers.**generate_key_data_from_nonce**(*server_nonce*, *new_nonce*)
> Generates the key data corresponding to the given nonce

telethon.helpers.**generate_random_long**(*signed=True*)
> Generates a random long integer (8 bytes), which is optionally signed

telethon.helpers.**retry_range**(*retries*)
> Generates an integer sequence starting from 1. If `retries` is not a zero or a positive integer value, the sequence
> will be infinite, otherwise it will end at `retries + 1`.

telethon.helpers.**strip_text**(*text*, *entities*)
> Strips whitespace from the given text modifying the provided entities.
>
> This assumes that there are no overlapping entities, that their length is greater or equal to one, and that their
> length is not out of bounds.

Symbols