pyTelegramBotAPI Documentation

Release 4.6.0

coder2020official

Jul 05, 2022

CONTENTS

CHAPTER

ONE

TELEBOT

TeleBot is synchronous and asynchronous implementation of Telegram Bot API.

1.1 Chats

English chat: Private chat

Russian chat: @pytelegrambotapi_talks_ru

News: @pyTelegramBotAPI

Pypi: Pypi

Source: Github repository

1.2 Some features:

Easy to learn and use.

Easy to understand.

Both sync and async.

Examples on features.

States

And more...

1.3 Content

1.3.1 Installation Guide

Using PIP

```
$ pip install pyTelegramBotAPI
```

Using pipenv

```
$ pipenv install pyTelegramBotAPI
```

By cloning repository

```
$ git clone https://github.com/eternnoir/pyTelegramBotAPI.git
$ cd pyTelegramBotAPI
$ python setup.py install
```

Directly using pip

```
$ pip install git+https://github.com/eternnoir/pyTelegramBotAPI.git
```

It is generally recommended to use the first option.

While the API is production-ready, it is still under development and it has regular updates, do not forget to update it regularly by calling:

```
$ pip install pytelegrambotapi --upgrade
```

1.3.2 Quick start

Synchronous TeleBot

```
#!/usr/bin/python

# This is a simple echo bot using the decorator mechanism.
# It echoes any incoming text messages.

import telebot

API_TOKEN = '<api_token>'

bot = telebot.TeleBot(API_TOKEN)


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


# Handle all other messages with content_type 'text' (content_types defaults to ['text'])
@bot.message_handler(func=lambda message: True)
```

(continues on next page)

(continued from previous page)

```
def echo_message(message):
    bot.reply_to(message, message.text)


bot.infinity_polling()
```

Asynchronous TeleBot

```
#!/usr/bin/python

# This is a simple echo bot using the decorator mechanism.
# It echoes any incoming text messages.

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



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


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


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

1.3.3 Types of API

class telebot.types.**Animation**(*file_id*, *file_unique_id*, *width=None*, *height=None*, *duration=None*, *thumb=None*, *file_name=None*, *mime_type=None*, *file_size=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Audio**(*file_id*, *file_unique_id*, *duration*, *performer=None*, *title=None*, *file_name=None*, *mime_type=None*, *file_size=None*, *thumb=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**BotCommand**(*command*, *description*)

Bases: *JsonSerializable*, *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**BotCommandScope**(*type='default'*, *chat_id=None*, *user_id=None*)

Bases: ABC, *JsonSerializable*

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**BotCommandScopeAllChatAdministrators**

Bases: *BotCommandScope*

class telebot.types.**BotCommandScopeAllGroupChats**

Bases: *BotCommandScope*

class telebot.types.**BotCommandScopeAllPrivateChats**

Bases: *BotCommandScope*

class telebot.types.**BotCommandScopeChat**(*chat_id=None*)

Bases: *BotCommandScope*

class telebot.types.**BotCommandScopeChatAdministrators**(*chat_id=None*)

Bases: *BotCommandScope*

class telebot.types.**BotCommandScopeChatMember**(*chat_id=None*, *user_id=None*)

Bases: *BotCommandScope*

class telebot.types.**BotCommandScopeDefault**

Bases: *BotCommandScope*

class telebot.types.**CallbackQuery**(*id*, *from_user*, *data*, *chat_instance*, *json_string*, *message=None*, *inline_message_id=None*, *game_short_name=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Chat**(*id*, *type*, *title=None*, *username=None*, *first_name=None*, *last_name=None*, *photo=None*, *bio=None*, *has_private_forwards=None*, *description=None*, *invite_link=None*, *pinned_message=None*, *permissions=None*, *slow_mode_delay=None*, *message_auto_delete_time=None*, *has_protected_content=None*, *sticker_set_name=None*, *can_set_sticker_set=None*, *linked_chat_id=None*, *location=None*, *join_to_send_messages=None*, *join_by_request=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**ChatAdministratorRights**(*is_anonymous: bool*, *can_manage_chat: bool*, *can_delete_messages: bool*, *can_manage_video_chats: bool*, *can_restrict_members: bool*, *can_promote_members: bool*, *can_change_info: bool*, *can_invite_users: bool*, *can_post_messages: Optional[bool] = None*, *can_edit_messages: Optional[bool] = None*, *can_pin_messages: Optional[bool] = None*)

Bases: *JsonDeserializable*, *JsonSerializable*

Class representation of: https://core.telegram.org/bots/api#chatadministratorrights

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**ChatInviteLink**(*invite_link*, *creator*, *creates_join_request*, *is_primary*, *is_revoked*, *name=None*, *expire_date=None*, *member_limit=None*, *pending_join_request_count=None*, ***kwargs*)

Bases: *JsonSerializable*, *JsonDeserializable*, *Dictionaryable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.ChatJoinRequest(*chat*, *from_user*, *date*, *bio=None*, *invite_link=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.ChatLocation(*location*, *address*, ***kwargs*)

Bases: JsonSerializable, JsonDeserializable, Dictionaryable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.ChatMember(*user*, *status*, *custom_title=None*, *is_anonymous=None*, *can_be_edited=None*, *can_post_messages=None*, *can_edit_messages=None*, *can_delete_messages=None*, *can_restrict_members=None*, *can_promote_members=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, *is_member=None*, *can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_manage_chat=None*, *can_manage_video_chats=None*, *until_date=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**ChatMemberAdministrator**(*user*, *status*, *custom_title=None*, *is_anonymous=None*, *can_be_edited=None*, *can_post_messages=None*, *can_edit_messages=None*, *can_delete_messages=None*, *can_restrict_members=None*, *can_promote_members=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, *is_member=None*, *can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_manage_chat=None*, *can_manage_video_chats=None*, *until_date=None*, ***kwargs*)

Bases: *ChatMember*

class telebot.types.**ChatMemberBanned**(*user*, *status*, *custom_title=None*, *is_anonymous=None*, *can_be_edited=None*, *can_post_messages=None*, *can_edit_messages=None*, *can_delete_messages=None*, *can_restrict_members=None*, *can_promote_members=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, *is_member=None*, *can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_manage_chat=None*, *can_manage_video_chats=None*, *until_date=None*, ***kwargs*)

Bases: *ChatMember*

class telebot.types.**ChatMemberLeft**(*user*, *status*, *custom_title=None*, *is_anonymous=None*, *can_be_edited=None*, *can_post_messages=None*, *can_edit_messages=None*, *can_delete_messages=None*, *can_restrict_members=None*, *can_promote_members=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, *is_member=None*, *can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_manage_chat=None*, *can_manage_video_chats=None*, *until_date=None*, ***kwargs*)

Bases: *ChatMember*

class telebot.types.**ChatMemberMember**(*user*, *status*, *custom_title=None*, *is_anonymous=None*, *can_be_edited=None*, *can_post_messages=None*, *can_edit_messages=None*, *can_delete_messages=None*, *can_restrict_members=None*, *can_promote_members=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, *is_member=None*, *can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_manage_chat=None*, *can_manage_video_chats=None*, *until_date=None*, ***kwargs*)

Bases: *ChatMember*

class telebot.types.**ChatMemberOwner**(*user*, *status*, *custom_title=None*, *is_anonymous=None*, *can_be_edited=None*, *can_post_messages=None*, *can_edit_messages=None*, *can_delete_messages=None*, *can_restrict_members=None*, *can_promote_members=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, *is_member=None*, *can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_manage_chat=None*, *can_manage_video_chats=None*, *until_date=None*, ***kwargs*)

Bases: *ChatMember*

class telebot.types.**ChatMemberRestricted**(*user*, *status*, *custom_title=None*, *is_anonymous=None*, *can_be_edited=None*, *can_post_messages=None*, *can_edit_messages=None*, *can_delete_messages=None*, *can_restrict_members=None*, *can_promote_members=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, *is_member=None*, *can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_manage_chat=None*, *can_manage_video_chats=None*, *until_date=None*, ***kwargs*)

Bases: *ChatMember*

class telebot.types.**ChatMemberUpdated**(*chat*, *from_user*, *date*, *old_chat_member*, *new_chat_member*, *invite_link=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

property difference: Dict[str, List]

Get the difference between *old_chat_member* and *new_chat_member* as a dict in the following format {'parameter': [old_value, new_value]} E.g {'status': ['member', 'kicked'], 'until_date': [None, 1625055092]}

class telebot.types.**ChatPermissions**(*can_send_messages=None*, *can_send_media_messages=None*, *can_send_polls=None*, *can_send_other_messages=None*, *can_add_web_page_previews=None*, *can_change_info=None*, *can_invite_users=None*, *can_pin_messages=None*, ***kwargs*)

Bases: *JsonDeserializable*, *JsonSerializable*, *Dictionaryable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**ChatPhoto**(*small_file_id*, *small_file_unique_id*, *big_file_id*, *big_file_unique_id*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**ChosenInlineResult**(*result_id*, *from_user*, *query*, *location=None*, *inline_message_id=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Contact**(*phone_number*, *first_name*, *last_name=None*, *user_id=None*, *vcard=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Dice**(*value*, *emoji*, ***kwargs*)

Bases: *JsonSerializable*, *Dictionaryable*, *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**Dictionaryable**

Bases: `object`

Subclasses of this class are guaranteed to be able to be converted to dictionary. All subclasses of this class must override to_dict.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**Document**(*file_id*, *file_unique_id*, *thumb=None*, *file_name=None*, *mime_type=None*, *file_size=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**File**(*file_id*, *file_unique_id*, *file_size=None*, *file_path=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**ForceReply**(*selective: Optional[bool] = None*, *input_field_placeholder: Optional[str] = None*)

Bases: JsonSerializable

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**Game**(*title*, *description*, *photo*, *text=None*, *text_entities=None*, *animation=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

classmethod parse_entities(*message_entity_array*)

classmethod parse_photo(*photo_size_array*)

class telebot.types.**GameHighScore**(*position*, *user*, *score*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**GroupChat**(*id*, *title*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**InlineKeyboardButton**(*text*, *url=None*, *callback_data=None*, *web_app=None*, *switch_inline_query=None*, *switch_inline_query_current_chat=None*, *callback_game=None*, *pay=None*, *login_url=None*, ***kwargs*)

Bases: *Dictionaryable*, *JsonSerializable*, *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**InlineKeyboardMarkup**(*keyboard=None*, *row_width=3*)

Bases: *Dictionaryable*, *JsonSerializable*, *JsonDeserializable*

add(**args*, *row_width=None*)

This method adds buttons to the keyboard without exceeding row_width.

E.g. InlineKeyboardMarkup.add("A", "B", "C") yields the json result: {keyboard: [["A"], ["B"], ["C"]]} when row_width is set to 1. When row_width is set to 2, the result: {keyboard: [["A", "B"], ["C"]]} See https://core.telegram.org/bots/api#inlinekeyboardmarkup

Parameters

- **args** – Array of InlineKeyboardButton to append to the keyboard
- **row_width** – width of row

Returns
self, to allow function chaining.

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

max_row_keys = 8

row(**args*)

Adds a list of InlineKeyboardButton to the keyboard. This method does not consider row_width.

InlineKeyboardMarkup.row("A").row("B", "C").to_json() outputs: '{keyboard: [["A"], ["B", "C"]]}' See https://core.telegram.org/bots/api#inlinekeyboardmarkup

Parameters
args – Array of InlineKeyboardButton to append to the keyboard

Returns
self, to allow function chaining.

`to_dict()`

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

`to_json()`

Converts this object to its json representation following the Telegram API guidelines described here: https://core.telegram.org/bots/api#inlinekeyboardmarkup :return:

`class telebot.types.InlineQuery(id, from_user, query, offset, chat_type=None, location=None, **kwargs)`

Bases: *JsonDeserializable*

`classmethod de_json(json_string)`

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

`class telebot.types.InlineQueryResultArticle(id, title, input_message_content, reply_markup=None, url=None, hide_url=None, description=None, thumb_url=None, thumb_width=None, thumb_height=None)`

Bases: *InlineQueryResultBase*

`to_dict()`

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

`class telebot.types.InlineQueryResultAudio(id, audio_url, title, caption=None, caption_entities=None, parse_mode=None, performer=None, audio_duration=None, reply_markup=None, input_message_content=None)`

Bases: *InlineQueryResultBase*

`to_dict()`

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

`class telebot.types.InlineQueryResultBase(type, id, title=None, caption=None, input_message_content=None, reply_markup=None, caption_entities=None, parse_mode=None)`

Bases: ABC, *Dictionaryable*, *JsonSerializable*

`to_dict()`

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

`to_json()`

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**InlineQueryResultCachedAudio**(*id*, *audio_file_id*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultCachedBase**

Bases: ABC, *JsonSerializable*

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**InlineQueryResultCachedDocument**(*id*, *document_file_id*, *title*, *description=None*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultCachedGif**(*id*, *gif_file_id*, *title=None*, *description=None*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultCachedMpeg4Gif**(*id*, *mpeg4_file_id*, *title=None*, *description=None*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultCachedPhoto**(*id*, *photo_file_id*, *title=None*, *description=None*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultCachedSticker**(*id*, *sticker_file_id*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultCachedVideo**(*id*, *video_file_id*, *title*, *description=None*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultCachedVoice**(*id*, *voice_file_id*, *title*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultCachedBase*

class telebot.types.**InlineQueryResultContact**(*id*, *phone_number*, *first_name*, *last_name=None*, *vcard=None*, *reply_markup=None*, *input_message_content=None*, *thumb_url=None*, *thumb_width=None*, *thumb_height=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultDocument**(*id*, *title*, *document_url*, *mime_type*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *description=None*, *reply_markup=None*, *input_message_content=None*, *thumb_url=None*, *thumb_width=None*, *thumb_height=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultGame**(*id*, *game_short_name*, *reply_markup=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultGif**(*id*, *gif_url*, *thumb_url*, *gif_width=None*, *gif_height=None*, *title=None*, *caption=None*, *caption_entities=None*, *reply_markup=None*, *input_message_content=None*, *gif_duration=None*, *parse_mode=None*, *thumb_mime_type=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultLocation**(*id*, *title*, *latitude*, *longitude*, *horizontal_accuracy*, *live_period=None*, *reply_markup=None*, *input_message_content=None*, *thumb_url=None*, *thumb_width=None*, *thumb_height=None*, *heading=None*, *proximity_alert_radius=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultMpeg4Gif**(*id*, *mpeg4_url*, *thumb_url*, *mpeg4_width=None*, *mpeg4_height=None*, *title=None*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*, *mpeg4_duration=None*, *thumb_mime_type=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultPhoto**(*id*, *photo_url*, *thumb_url*, *photo_width=None*, *photo_height=None*, *title=None*, *description=None*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultVenue**(*id*, *title*, *latitude*, *longitude*, *address*, *foursquare_id=None*, *foursquare_type=None*, *reply_markup=None*, *input_message_content=None*, *thumb_url=None*, *thumb_width=None*, *thumb_height=None*, *google_place_id=None*, *google_place_type=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultVideo**(*id*, *video_url*, *mime_type*, *thumb_url*, *title*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *video_width=None*, *video_height=None*, *video_duration=None*, *description=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InlineQueryResultVoice**(*id*, *voice_url*, *title*, *caption=None*, *caption_entities=None*, *parse_mode=None*, *voice_duration=None*, *reply_markup=None*, *input_message_content=None*)

Bases: *InlineQueryResultBase*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputContactMessageContent**(*phone_number*, *first_name*, *last_name=None*, *vcard=None*)

Bases: *Dictionaryable*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputInvoiceMessageContent**(*title*, *description*, *payload*, *provider_token*, *currency*, *prices*, *max_tip_amount=None*, *suggested_tip_amounts=None*, *provider_data=None*, *photo_url=None*, *photo_size=None*, *photo_width=None*, *photo_height=None*, *need_name=None*, *need_phone_number=None*, *need_email=None*, *need_shipping_address=None*, *send_phone_number_to_provider=None*, *send_email_to_provider=None*, *is_flexible=None*)

Bases: *Dictionaryable*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputLocationMessageContent**(*latitude*, *longitude*, *horizontal_accuracy=None*, *live_period=None*, *heading=None*, *proximity_alert_radius=None*)

Bases: *Dictionaryable*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputMedia**(*type*, *media*, *caption=None*, *parse_mode=None*, *caption_entities=None*)

Bases: *Dictionaryable*, *JsonSerializable*

convert_input_media()

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**InputMediaAnimation**(*media*, *thumb=None*, *caption=None*, *parse_mode=None*, *width=None*, *height=None*, *duration=None*)

Bases: *InputMedia*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputMediaAudio**(*media*, *thumb=None*, *caption=None*, *parse_mode=None*, *duration=None*, *performer=None*, *title=None*)

Bases: *InputMedia*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputMediaDocument**(*media*, *thumb=None*, *caption=None*, *parse_mode=None*, *disable_content_type_detection=None*)

Bases: *InputMedia*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputMediaPhoto**(*media*, *caption=None*, *parse_mode=None*)

Bases: *InputMedia*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputMediaVideo**(*media*, *thumb=None*, *caption=None*, *parse_mode=None*, *width=None*, *height=None*, *duration=None*, *supports_streaming=None*)

Bases: *InputMedia*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputTextMessageContent**(*message_text*, *parse_mode=None*, *entities=None*, *disable_web_page_preview=None*)

Bases: *Dictionaryable*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**InputVenueMessageContent**(*latitude*, *longitude*, *title*, *address*, *foursquare_id=None*, *foursquare_type=None*, *google_place_id=None*, *google_place_type=None*)

Bases: *Dictionaryable*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**Invoice**(*title*, *description*, *start_parameter*, *currency*, *total_amount*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**JsonDeserializable**

Bases: object

Subclasses of this class are guaranteed to be able to be created from a json-style dict or json formatted string. All subclasses of this class must override de_json.

static **check_json**(*json_type*, *dict_copy=True*)

Checks whether json_type is a dict or a string. If it is already a dict, it is returned as-is. If it is not, it is converted to a dict by means of json.loads(json_type)

Parameters

- **json_type** – input json or parsed dict
- **dict_copy** – if dict is passed and it is changed outside - should be True!

Returns
Dictionary parsed from json or original dict

classmethod **de_json**(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**JsonSerializable**

Bases: object

Subclasses of this class are guaranteed to be able to be converted to JSON format. All subclasses of this class must override to_json.

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**KeyboardButton**(*text: str*, *request_contact: Optional[bool] = None*, *request_location: Optional[bool] = None*, *request_poll: Optional[KeyboardButtonPollType] = None*, *web_app: Optional[WebAppInfo] = None*)

Bases: *Dictionaryable*, *JsonSerializable*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**KeyboardButtonPollType**(*type=''*)

Bases: *Dictionaryable*

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

class telebot.types.**LabeledPrice**(*label*, *amount*)

Bases: *JsonSerializable*

to_dict()

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**Location**(*longitude*, *latitude*, *horizontal_accuracy=None*, *live_period=None*, *heading=None*, *proximity_alert_radius=None*, ***kwargs*)

Bases: *JsonDeserializable*, *JsonSerializable*, *Dictionaryable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**LoginUrl**(*url*, *forward_text=None*, *bot_username=None*, *request_write_access=None*, ***kwargs*)

Bases: *Dictionaryable*, *JsonSerializable*, *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**MaskPosition**(*point*, *x_shift*, *y_shift*, *scale*, ***kwargs*)

Bases: *Dictionaryable*, *JsonDeserializable*, *JsonSerializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**MenuButton**

Bases: *JsonDeserializable*, *JsonSerializable*

Base class for MenuButtons.

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**MenuButtonCommands**(*type*)

Bases: *MenuButton*

to_dict()

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**MenuButtonDefault**(*type*)

Bases: *MenuButton*

to_dict()

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**MenuButtonWebApp**(*type*, *text*, *web_app*)

Bases: *MenuButton*

to_dict()

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**Message**(*message_id*, *from_user*, *date*, *chat*, *content_type*, *options*, *json_string*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

property html_caption

property html_text

classmethod parse_chat(*chat*)

classmethod parse_entities(*message_entity_array*)

classmethod parse_photo(*photo_size_array*)

class telebot.types.**MessageAutoDeleteTimerChanged**(*message_auto_delete_time*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**MessageEntity**(*type*, *offset*, *length*, *url=None*, *user=None*, *language=None*, ***kwargs*)

Bases: *Dictionaryable*, *JsonSerializable*, *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

static to_list_of_dicts(*entity_list*) → Optional[List[Dict]]

class telebot.types.**MessageID**(*message_id*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**OrderInfo**(*name=None*, *phone_number=None*, *email=None*, *shipping_address=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**PhotoSize**(*file_id*, *file_unique_id*, *width*, *height*, *file_size=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Poll**(*question*, *options*, *poll_id=None*, *total_voter_count=None*, *is_closed=None*, *is_anonymous=None*, *type=None*, *allows_multiple_answers=None*, *correct_option_id=None*, *explanation=None*, *explanation_entities=None*, *open_period=None*, *close_date=None*, *poll_type=None*, ***kwargs*)

Bases: *JsonDeserializable*

add(*option*)

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**PollAnswer**(*poll_id*, *user*, *option_ids*, ***kwargs*)

Bases: *JsonSerializable*, *JsonDeserializable*, *Dictionaryable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**PollOption**(*text*, *voter_count=0*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**PreCheckoutQuery**(*id*, *from_user*, *currency*, *total_amount*, *invoice_payload*, *shipping_option_id=None*, *order_info=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**ProximityAlertTriggered**(*traveler*, *watcher*, *distance*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**ReplyKeyboardMarkup**(*resize_keyboard: Optional[bool] = None*, *one_time_keyboard: Optional[bool] = None*, *selective: Optional[bool] = None*, *row_width: int = 3*, *input_field_placeholder: Optional[str] = None*)

Bases: *JsonSerializable*

add(**args*, *row_width=None*)

This function adds strings to the keyboard, while not exceeding row_width. E.g. ReplyKeyboardMarkup#add("A", "B", "C") yields the json result {keyboard: [["A"], ["B"], ["C"]]} when row_width is set to 1. When row_width is set to 2, the following is the result of this function: {keyboard: [["A", "B"], ["C"]]} See https://core.telegram.org/bots/api#replykeyboardmarkup

Parameters

- **args** – KeyboardButton to append to the keyboard
- **row_width** – width of row

Returns
self, to allow function chaining.

max_row_keys = 12

row(**args*)

Adds a list of KeyboardButton to the keyboard. This function does not consider row_width. ReplyKeyboardMarkup#row("A")#row("B", "C")#to_json() outputs '{keyboard: [["A"], ["B", "C"]]}' See https://core.telegram.org/bots/api#replykeyboardmarkup

Parameters
args – strings

Returns
self, to allow function chaining.

to_json()

Converts this object to its json representation following the Telegram API guidelines described here: https://core.telegram.org/bots/api#replykeyboardmarkup :return:

class telebot.types.**ReplyKeyboardRemove**(*selective=None*)

Bases: *JsonSerializable*

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**SentWebAppMessage**(*inline_message_id=None*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

class telebot.types.**ShippingAddress**(*country_code*, *state*, *city*, *street_line1*, *street_line2*, *post_code*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**ShippingOption**(*id*, *title*)

Bases: *JsonSerializable*

add_price(**args*)

Add LabeledPrice to ShippingOption

Parameters
args – LabeledPrices

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**ShippingQuery**(*id*, *from_user*, *invoice_payload*, *shipping_address*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Sticker**(*file_id*, *file_unique_id*, *width*, *height*, *is_animated*, *is_video*, *thumb=None*, *emoji=None*, *set_name=None*, *mask_position=None*, *file_size=None*, *premium_animation=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**StickerSet**(*name*, *title*, *is_animated*, *is_video*, *contains_masks*, *stickers*, *thumb=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**SuccessfulPayment**(*currency*, *total_amount*, *invoice_payload*, *shipping_option_id=None*, *order_info=None*, *telegram_payment_charge_id=None*, *provider_payment_charge_id=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Update**(*update_id*, *message*, *edited_message*, *channel_post*, *edited_channel_post*, *inline_query*, *chosen_inline_result*, *callback_query*, *shipping_query*, *pre_checkout_query*, *poll*, *poll_answer*, *my_chat_member*, *chat_member*, *chat_join_request*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**User**(*id*, *is_bot*, *first_name*, *last_name=None*, *username=None*, *language_code=None*, *can_join_groups=None*, *can_read_all_group_messages=None*, *supports_inline_queries=None*, *is_premium=None*, *added_to_attachment_menu=None*, ***kwargs*)

Bases: *JsonDeserializable*, *Dictionaryable*, *JsonSerializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

property full_name

to_dict()

Returns a DICT with class field values

This function must be overridden by subclasses. :return: a DICT

to_json()

Returns a JSON string representation of this class.

This function must be overridden by subclasses. :return: a JSON formatted string.

class telebot.types.**UserProfilePhotos**(*total_count*, *photos=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Venue**(*location*, *title*, *address*, *foursquare_id=None*, *foursquare_type=None*, *google_place_id=None*, *google_place_type=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Video**(*file_id*, *file_unique_id*, *width*, *height*, *duration*, *thumb=None*, *file_name=None*, *mime_type=None*, *file_size=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**VideoChatEnded**(*duration*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**VideoChatParticipantsInvited**(*users=None*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**VideoChatScheduled**(*start_date*, ***kwargs*)

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**VideoChatStarted**

Bases: JsonDeserializable

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**VideoNote**(*file_id*, *file_unique_id*, *length*, *duration*, *thumb=None*, *file_size=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**Voice**(*file_id*, *file_unique_id*, *duration*, *mime_type=None*, *file_size=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

class telebot.types.**VoiceChatEnded**(**args*, ***kwargs*)

Bases: *VideoChatEnded*

class telebot.types.**VoiceChatParticipantsInvited**(**args*, ***kwargs*)

Bases: *VideoChatParticipantsInvited*

class telebot.types.**VoiceChatScheduled**(**args*, ***kwargs*)

Bases: *VideoChatScheduled*

class telebot.types.**VoiceChatStarted**

Bases: *VideoChatStarted*

class telebot.types.**WebAppData**(*data*, *button_text*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

class telebot.types.**WebAppInfo**(*url*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

to_dict()

class telebot.types.**WebhookInfo**(*url*, *has_custom_certificate*, *pending_update_count*, *ip_address=None*, *last_error_date=None*, *last_error_message=None*, *last_synchronization_error_date=None*, *max_connections=None*, *allowed_updates=None*, ***kwargs*)

Bases: *JsonDeserializable*

classmethod de_json(*json_string*)

Returns an instance of this class from the given json dict or string.

This function must be overridden by subclasses. :return: an instance of this class created from the given json dict or string.

1.3.4 TeleBot version

TeleBot methods

class telebot.**ExceptionHandler**

Bases: object

Class for handling exceptions while Polling

handle(*exception*)

class telebot.**Handler**(*callback*, **args*, ***kwargs*)

Bases: object

Class for (next step|reply) handlers

telebot.**REPLY_MARKUP_TYPES**

telebot

Type
Module

alias of Union[*InlineKeyboardMarkup*, *ReplyKeyboardMarkup*, *ReplyKeyboardRemove*, *ForceReply*]

class telebot.**TeleBot**(*token*, *parse_mode=None*, *threaded=True*, *skip_pending=False*, *num_threads=2*, *next_step_backend=None*, *reply_backend=None*, *exception_handler=None*, *last_update_id=0*, *suppress_middleware_excepions=False*, *state_storage=<telebot.storage.memory_storage.StateMemoryStorage object>*, *use_class_middlewares=False*)

Bases: object

This is the main synchronous class for Bot.

It allows you to add handlers for different kind of updates.

Usage:

```
from telebot import TeleBot
bot = TeleBot('token') # get token from @BotFather
```

See more examples in examples/ directory: https://github.com/eternnoir/pyTelegramBotAPI/tree/master/examples

add_callback_query_handler(*handler_dict*)

Adds a callback request handler Note that you should use register_callback_query_handler to add callback_query_handler to the bot.

Parameters
handler_dict –

Returns

add_channel_post_handler(*handler_dict*)

Adds channel post handler Note that you should use register_channel_post_handler to add channel_post_handler to the bot.

Parameters
handler_dict –

Returns

add_chat_join_request_handler(*handler_dict*)

Adds a chat_join_request handler. Note that you should use register_chat_join_request_handler to add chat_join_request_handler to the bot.

Parameters
handler_dict –

Returns

add_chat_member_handler(*handler_dict*)

Adds a chat_member handler. Note that you should use register_chat_member_handler to add chat_member_handler to the bot.

Parameters
handler_dict –

Returns

add_chosen_inline_handler(*handler_dict*)

Description: TBD Note that you should use register_chosen_inline_handler to add chosen_inline_handler to the bot.

Parameters
handler_dict –

Returns

add_custom_filter(*custom_filter: Union[*SimpleCustomFilter, AdvancedCustomFilter*]*)

Create custom filter.

Parameters

- **custom_filter** – Class with check(message) method.
- **custom_filter** – Custom filter class with key.

add_data(*user_id: int*, *chat_id: Optional[int] = None*, ***kwargs*)

Add data to states.

Parameters

- **user_id** –
- **chat_id** –

add_edited_channel_post_handler(*handler_dict*)

Adds the edit channel post handler Note that you should use register_edited_channel_post_handler to add edited_channel_post_handler to the bot.

Parameters
handler_dict –

Returns

add_edited_message_handler(*handler_dict*)

Adds the edit message handler Note that you should use register_edited_message_handler to add edited_message_handler to the bot.

Parameters
handler_dict –

Returns

add_inline_handler(*handler_dict*)

Adds inline call handler Note that you should use register_inline_handler to add inline_handler to the bot.

Parameters
handler_dict –

Returns

add_message_handler(*handler_dict*)

Adds a message handler Note that you should use register_message_handler to add message_handler to the bot.

Parameters
handler_dict –

Returns

add_middleware_handler(*handler*, *update_types=None*)

Add middleware handler.

Parameters

- handler –
- update_types –

Returns

add_my_chat_member_handler(*handler_dict*)

Adds a my_chat_member handler. Note that you should use register_my_chat_member_handler to add my_chat_member_handler to the bot.

Parameters
handler_dict –

Returns

add_poll_answer_handler(*handler_dict*)

Adds a poll_answer request handler. Note that you should use register_poll_answer_handler to add poll_answer_handler to the bot.

Parameters
handler_dict –

Returns

add_poll_handler(*handler_dict*)

Adds a poll request handler Note that you should use register_poll_handler to add poll_handler to the bot.

Parameters
handler_dict –

Returns

add_pre_checkout_query_handler(*handler_dict*)

Adds a pre-checkout request handler Note that you should use register_pre_checkout_query_handler to add pre_checkout_query_handler to the bot.

Parameters
handler_dict –

Returns

add_shipping_query_handler(*handler_dict*)

Adds a shipping request handler. Note that you should use register_shipping_query_handler to add shipping_query_handler to the bot.

Parameters
handler_dict –

Returns

add_sticker_to_set(*user_id: int*, *name: str*, *emojis: str*, *png_sticker: Optional[Union[Any, str]] = None*, *tgs_sticker: Optional[Union[Any, str]] = None*, *webm_sticker: Optional[Union[Any, str]] = None*, *mask_position: Optional[*MaskPosition*] = None*) → bool

Use this method to add a new sticker to a set created by the bot. It's required to pass *png_sticker* or *tgs_sticker*. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#addstickertoset

Parameters

- **user_id** –
- **name** –
- **emojis** –
- **png_sticker** – Required if *tgs_sticker* is None
- **tgs_sticker** – Required if *png_sticker* is None
- **webm_sticker** –
- **mask_position** –

Returns

answer_callback_query(*callback_query_id: int*, *text: Optional[str] = None*, *show_alert: Optional[bool] = None*, *url: Optional[str] = None*, *cache_time: Optional[int] = None*) → bool

Use this method to send answers to callback queries sent from inline keyboards. The answer will be displayed to the user as a notification at the top of the chat screen or as an alert.

Telegram documentation: https://core.telegram.org/bots/api#answercallbackquery

Parameters

- **callback_query_id** –
- **text** –
- **show_alert** –
- **url** –
- **cache_time** –

Returns

answer_inline_query(*inline_query_id: str*, *results: List[Any]*, *cache_time: Optional[int] = None*, *is_personal: Optional[bool] = None*, *next_offset: Optional[str] = None*, *switch_pm_text: Optional[str] = None*, *switch_pm_parameter: Optional[str] = None*) → bool

Use this method to send answers to an inline query. On success, True is returned. No more than 50 results per query are allowed.

Telegram documentation: https://core.telegram.org/bots/api#answerinlinequery

Parameters

- **inline_query_id** – Unique identifier for the answered query
- **results** – Array of results for the inline query
- **cache_time** – The maximum amount of time in seconds that the result of the inline query may be cached on the server.
- **is_personal** – Pass True, if results may be cached on the server side only for the user that sent the query.
- **next_offset** – Pass the offset that a client should send in the next query with the same text to receive more results.
- **switch_pm_parameter** – If passed, clients will display a button with specified text that switches the user to a private chat with the bot and sends the bot a start message with the parameter switch_pm_parameter
- **switch_pm_text** – Parameter for the start message sent to the bot when user presses the switch button

Returns

True means success.

answer_pre_checkout_query(*pre_checkout_query_id: int*, *ok: bool*, *error_message: Optional[str] = None*) → bool

Response to a request for pre-inspection.

Telegram documentation: https://core.telegram.org/bots/api#answerprecheckoutquery

Parameters

- **pre_checkout_query_id** –
- **ok** –
- **error_message** –

Returns

answer_shipping_query(*shipping_query_id: str*, *ok: bool*, *shipping_options: Optional[List[ShippingOption]] = None*, *error_message: Optional[str] = None*) → bool

Asks for an answer to a shipping question.

Telegram documentation: https://core.telegram.org/bots/api#answershippingquery

Parameters

- **shipping_query_id** –
- **ok** –
- **shipping_options** –

- **error_message** –

Returns

answer_web_app_query(*web_app_query_id: str*, *result:* InlineQueryResultBase) → *SentWebAppMessage*

Use this method to set the result of an interaction with a Web App and send a corresponding message on behalf of the user to the chat from which the query originated. On success, a SentWebAppMessage object is returned.

Telegram Documentation: https://core.telegram.org/bots/api#answerwebappquery

Parameters

- **web_app_query_id** – Unique identifier for the query to be answered
- **result** – A JSON-serialized object describing the message to be sent

Returns

approve_chat_join_request(*chat_id: Union[str, int]*, *user_id: Union[int, str]*) → bool

Use this method to approve a chat join request. The bot must be an administrator in the chat for this to work and must have the can_invite_users administrator right. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#approvechatjoinrequest

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **user_id** – Unique identifier of the target user

Returns
True on success.

ban_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *until_date: Optional[Union[int, datetime]] = None*, *revoke_messages: Optional[bool] = None*) → bool

Use this method to ban a user in a group, a supergroup or a channel. In the case of supergroups and channels, the user will not be able to return to the chat on their own using invite links, etc., unless unbanned first. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#banchatmember

Parameters

- **chat_id** – Int or string : Unique identifier for the target group or username of the target supergroup
- **user_id** – Int : Unique identifier of the target user
- **until_date** – Date when the user will be unbanned, unix time. If user is banned for more than 366 days or less than 30 seconds from the current time they are considered to be banned forever
- **revoke_messages** – Bool: Pass True to delete all messages from the chat for the user that is being removed. If False, the user will be able to see messages in the group that were sent before the user was removed. Always True for supergroups and channels.

Returns
boolean

ban_chat_sender_chat(*chat_id: Union[int, str]*, *sender_chat_id: Union[int, str]*) → bool

Use this method to ban a channel chat in a supergroup or a channel. The owner of the chat will not be able to send messages and join live streams on behalf of the chat, unless it is unbanned first. The bot must be an administrator in the supergroup or channel for this to work and must have the appropriate administrator rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#banchatsenderchat

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **sender_chat_id** – Unique identifier of the target sender chat

Returns

True on success.

callback_query_handler(*func*, ***kwargs*)

Callback request handler decorator

Parameters

- **func** –
- **kwargs** –

Returns

channel_post_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, ***kwargs*)

Channel post handler decorator

Parameters

- **commands** –
- **regexp** –
- **func** –
- **content_types** –
- **kwargs** –

Returns

chat_join_request_handler(*func=None*, ***kwargs*)

chat_join_request handler

Parameters

- **func** –
- **kwargs** –

Returns

chat_member_handler(*func=None*, ***kwargs*)

chat_member handler.

Parameters

- **func** –
- **kwargs** –

Returns

static check_commands_input(*commands*, *method_name*)

static check_regexp_input(*regexp*, *method_name*)

chosen_inline_handler(*func*, ***kwargs*)

Description: TBD

Parameters

- **func** –
- **kwargs** –

Returns

clear_reply_handlers(*message: Message*) → None

Clears all callback functions registered by register_for_reply() and register_for_reply_by_message_id().

Parameters

message – The message for which we want to clear reply handlers

clear_reply_handlers_by_message_id(*message_id: int*) → None

Clears all callback functions registered by register_for_reply() and register_for_reply_by_message_id().

Parameters

message_id – The message id for which we want to clear reply handlers

clear_step_handler(*message: Message*) → None

Clears all callback functions registered by register_next_step_handler().

Parameters

message – The message for which we want to handle new message after that in same chat.

clear_step_handler_by_chat_id(*chat_id: Union[int, str]*) → None

Clears all callback functions registered by register_next_step_handler().

Parameters

chat_id – The chat for which we want to clear next step handlers

close() → bool

Use this method to close the bot instance before moving it from one local server to another. You need to delete the webhook before calling this method to ensure that the bot isn't launched again after server restart. The method will return error 429 in the first 10 minutes after the bot is launched. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#close

copy_message(*chat_id: Union[int, str]*, *from_chat_id: Union[int, str]*, *message_id: int*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[MessageEntity]] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *timeout: Optional[int] = None*) → *MessageID*

Use this method to copy messages of any kind.

Telegram documentation: https://core.telegram.org/bots/api#copymessage

Parameters

- **chat_id** – which chat to forward
- **from_chat_id** – which chat message from

- **message_id** – message id
- **caption** –
- **parse_mode** –
- **caption_entities** –
- **disable_notification** –
- **protect_content** –
- **reply_to_message_id** –
- **allow_sending_without_reply** –
- **reply_markup** –
- **timeout** –

Returns
API reply.

create_chat_invite_link(*chat_id: Union[int, str]*, *name: Optional[str] = None*, *expire_date: Optional[Union[int, datetime]] = None*, *member_limit: Optional[int] = None*, *creates_join_request: Optional[bool] = None*) → *ChatInviteLink*

Use this method to create an additional invite link for a chat. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#createchatinvitelink

Parameters

- **chat_id** – Id: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **name** – Invite link name; 0-32 characters
- **expire_date** – Point in time (Unix timestamp) when the link will expire
- **member_limit** – Maximum number of users that can be members of the chat simultaneously
- **creates_join_request** – True, if users joining the chat via the link need to be approved by chat administrators. If True, member_limit can't be specified

Returns

create_invoice_link(*title: str*, *description: str*, *payload: str*, *provider_token: str*, *currency: str*, *prices: List[LabeledPrice]*, *max_tip_amount: Optional[int] = None*, *suggested_tip_amounts: Optional[List[int]] = None*, *provider_data: Optional[str] = None*, *photo_url: Optional[str] = None*, *photo_size: Optional[int] = None*, *photo_width: Optional[int] = None*, *photo_height: Optional[int] = None*, *need_name: Optional[bool] = None*, *need_phone_number: Optional[bool] = None*, *need_email: Optional[bool] = None*, *need_shipping_address: Optional[bool] = None*, *send_phone_number_to_provider: Optional[bool] = None*, *send_email_to_provider: Optional[bool] = None*, *is_flexible: Optional[bool] = None*) → str

Use this method to create a link for an invoice. Returns the created invoice link as String on success.

Telegram documentation: https://core.telegram.org/bots/api#createinvoicelink

Parameters

- **title** – Product name, 1-32 characters

- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use for your internal processes.
- **provider_token** – Payments provider token, obtained via @Botfather
- **currency** – Three-letter ISO 4217 currency code, see https://core.telegram.org/bots/payments#supported-currencies
- **prices** – Price breakdown, a list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.)
- **max_tip_amount** – The maximum accepted amount for tips in the smallest units of the currency
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the smallest
- **provider_data** – A JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass True, if you require the user's full name to complete the order
- **need_phone_number** – Pass True, if you require the user's phone number to complete the order
- **need_email** – Pass True, if you require the user's email to complete the order
- **need_shipping_address** – Pass True, if you require the user's shipping address to complete the order
- **send_phone_number_to_provider** – Pass True, if user's phone number should be sent to provider
- **send_email_to_provider** – Pass True, if user's email address should be sent to provider
- **is_flexible** – Pass True, if the final price depends on the shipping method

Returns

Created invoice link as String on success.

create_new_sticker_set(*user_id: int*, *name: str*, *title: str*, *emojis: str*, *png_sticker: Optional[Union[Any, str]] = None*, *tgs_sticker: Optional[Union[Any, str]] = None*, *webm_sticker: Optional[Union[Any, str]] = None*, *contains_masks: Optional[bool] = None*, *mask_position: Optional[*MaskPosition*] = None*) → bool

Use this method to create new sticker set owned by a user. The bot will be able to edit the created sticker set. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#createnewstickerset

Parameters

- **user_id** –
- **name** –

- **title** –
- **emojis** –
- **png_sticker** –
- **tgs_sticker** –
- **webm_sticker** –
- **contains_masks** –
- **mask_position** –

Returns

decline_chat_join_request(*chat_id: Union[str, int]*, *user_id: Union[int, str]*) → bool

Use this method to decline a chat join request. The bot must be an administrator in the chat for this to work and must have the can_invite_users administrator right. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#declinechatjoinrequest

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **user_id** – Unique identifier of the target user

Returns

True on success.

delete_chat_photo(*chat_id: Union[int, str]*) → bool

Use this method to delete a chat photo. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success. Note: In regular groups (non-supergroups), this method will only work if the 'All Members Are Admins' setting is off in the target group.

Telegram documentation: https://core.telegram.org/bots/api#deletechatphoto

Parameters

chat_id – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)

delete_chat_sticker_set(*chat_id: Union[int, str]*) → bool

Use this method to delete a group sticker set from a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Use the field can_set_sticker_set optionally returned in getChat requests to check if the bot can use this method. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletechatstickerset

Parameters

chat_id – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)

Returns

API reply.

delete_message(*chat_id: Union[int, str]*, *message_id: int*, *timeout: Optional[int] = None*) → bool

Use this method to delete message. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletemessage

Parameters

- **chat_id** – in which chat to delete
- **message_id** – which message to delete
- **timeout** –

Returns
API reply.

delete_my_commands(*scope: Optional[BotCommandScope] = None*, *language_code: Optional[str] = None*) → bool

Use this method to delete the list of the bot's commands for the given scope and user language. After deletion, higher level commands will be shown to affected users. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletemycommands

Parameters

- **scope** – The scope of users for which the commands are relevant. Defaults to BotCommandScopeDefault.
- **language_code** – A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

delete_state(*user_id: int*, *chat_id: Optional[int] = None*) → None

Delete the current state of a user.

Parameters

- **user_id** –
- **chat_id** –

Returns

delete_sticker_from_set(*sticker: str*) → bool

Use this method to delete a sticker from a set created by the bot. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletestickerfromset

Parameters
sticker –

Returns

delete_webhook(*drop_pending_updates=None*, *timeout=None*)

Use this method to remove webhook integration if you decide to switch back to getUpdates.

Telegram documentation: https://core.telegram.org/bots/api#deletewebhook

Parameters

- **drop_pending_updates** – Pass True to drop all pending updates
- **timeout** – Integer. Request connection timeout

Returns
bool

disable_save_next_step_handlers()

Disable saving next step handlers (by default saving disable)

This function is left to keep backward compatibility whose purpose was to disable file saving capability for handlers. For the same purpose, MemoryHandlerBackend is reassigned as a new next_step_backend backend instead of FileHandlerBackend.

disable_save_reply_handlers()

Disable saving next step handlers (by default saving disable)

This function is left to keep backward compatibility whose purpose was to disable file saving capability for handlers. For the same purpose, MemoryHandlerBackend is reassigned as a new reply_backend backend instead of FileHandlerBackend.

download_file(*file_path: str*) → bytes

edit_chat_invite_link(*chat_id: Union[int, str]*, *invite_link: Optional[str] = None*, *name: Optional[str] = None*, *expire_date: Optional[Union[int, datetime]] = None*, *member_limit: Optional[int] = None*, *creates_join_request: Optional[bool] = None*) → *ChatInviteLink*

Use this method to edit a non-primary invite link created by the bot. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#editchatinvitelink

Parameters

- **chat_id** – Id: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **name** – Invite link name; 0-32 characters
- **invite_link** – The invite link to edit
- **expire_date** – Point in time (Unix timestamp) when the link will expire
- **member_limit** – Maximum number of users that can be members of the chat simultaneously
- **creates_join_request** – True, if users joining the chat via the link need to be approved by chat administrators. If True, member_limit can't be specified

Returns

edit_message_caption(*caption: str*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[MessageEntity]] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*) → Union[*Message*, bool]

Use this method to edit captions of messages.

Telegram documentation: https://core.telegram.org/bots/api#editmessagecaption

Parameters

- **caption** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **parse_mode** –
- **caption_entities** –

- **reply_markup** –

Returns

edit_message_live_location(*latitude: float*, *longitude: float*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*) → *Message*

Use this method to edit live location.

Telegram documentation: https://core.telegram.org/bots/api#editmessagelivelocation

Parameters

- **latitude** –
- **longitude** –
- **chat_id** –
- **message_id** –
- **reply_markup** –
- **timeout** –
- **inline_message_id** –
- **horizontal_accuracy** –
- **heading** –
- **proximity_alert_radius** –

Returns

edit_message_media(*media: Any*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*) → Union[*Message*, bool]

Use this method to edit animation, audio, document, photo, or video messages. If a message is a part of a message album, then it can be edited only to a photo or a video. Otherwise, message type can be changed arbitrarily. When inline message is edited, new file can't be uploaded. Use previously uploaded file via its file_id or specify a URL.

Telegram documentation: https://core.telegram.org/bots/api#editmessagemedia

Parameters

- **media** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **reply_markup** –

Returns

edit_message_reply_markup(*chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*) → Union[*Message*, bool]

Use this method to edit only the reply markup of messages.

Telegram documentation: https://core.telegram.org/bots/api#editmessagereplymarkup

Parameters

- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **reply_markup** –

Returns

edit_message_text(*text: str*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *parse_mode: Optional[str] = None*, *entities: Optional[List[*MessageEntity*]] = None*, *disable_web_page_preview: Optional[bool] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*) → Union[*Message*, bool]

Use this method to edit text and game messages.

Telegram documentation: https://core.telegram.org/bots/api#editmessagetext

Parameters

- **text** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **parse_mode** –
- **entities** –
- **disable_web_page_preview** –
- **reply_markup** –

Returns

edited_channel_post_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, ***kwargs*)

Edit channel post handler decorator

Parameters

- **commands** –
- **regexp** –
- **func** –
- **content_types** –
- **kwargs** –

Returns

edited_message_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, *chat_types=None*, ***kwargs*)

Edit message handler decorator

Parameters

- **commands** –
- **regexp** –
- **func** –
- **content_types** –
- **chat_types** – list of chat types
- **kwargs** –

Returns

enable_save_next_step_handlers(*delay=120*, *filename='./.handler-saves/step.save'*)

Enable saving next step handlers (by default saving disabled)

This function explicitly assigns FileHandlerBackend (instead of Saver) just to keep backward compatibility whose purpose was to enable file saving capability for handlers. And the same implementation is now available with FileHandlerBackend

Parameters

- **delay** – Delay between changes in handlers and saving
- **filename** – Filename of save file

enable_save_reply_handlers(*delay=120*, *filename='./.handler-saves/reply.save'*)

Enable saving reply handlers (by default saving disable)

This function explicitly assigns FileHandlerBackend (instead of Saver) just to keep backward compatibility whose purpose was to enable file saving capability for handlers. And the same implementation is now available with FileHandlerBackend

Parameters

- **delay** – Delay between changes in handlers and saving
- **filename** – Filename of save file

enable_saving_states(*filename='./.state-save/states.pkl'*)

Enable saving states (by default saving disabled)

Parameters
filename – Filename of saving file

export_chat_invite_link(*chat_id: Union[int, str]*) → str

Use this method to export an invite link to a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#exportchatinvitelink

Parameters
chat_id – Id: Unique identifier for the target chat or username of the target channel (in the format @channelusername)

Returns
exported invite link as String on success.

forward_message(*chat_id: Union[int, str]*, *from_chat_id: Union[int, str]*, *message_id: int*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to forward messages of any kind.

Telegram documentation: https://core.telegram.org/bots/api#forwardmessage

Parameters

- **disable_notification** –
- **chat_id** – which chat to forward
- **from_chat_id** – which chat message from
- **message_id** – message id
- **protect_content** – Protects the contents of the forwarded message from forwarding and saving
- **timeout** –

Returns
API reply.

get_chat(*chat_id: Union[int, str]*) → *Chat*

Use this method to get up to date information about the chat (current name of the user for one-on-one conversations, current username of a user, group or channel, etc.). Returns a Chat object on success.

Telegram documentation: https://core.telegram.org/bots/api#getchat

Parameters
chat_id –

Returns
API reply.

get_chat_administrators(*chat_id: Union[int, str]*) → List[*ChatMember*]

Use this method to get a list of administrators in a chat. On success, returns an Array of ChatMember objects that contains information about all chat administrators except other bots.

Telegram documentation: https://core.telegram.org/bots/api#getchatadministrators

Parameters
chat_id – Unique identifier for the target chat or username of the target supergroup or channel (in the format @channelusername)

Returns
API reply.

get_chat_member(*chat_id: Union[int, str]*, *user_id: int*) → *ChatMember*

Use this method to get information about a member of a chat. Returns a ChatMember object on success.

Telegram documentation: https://core.telegram.org/bots/api#getchatmember

Parameters

- **chat_id** –
- **user_id** –

Returns
API reply.

get_chat_member_count(*chat_id: Union[int, str]*) → int

Use this method to get the number of members in a chat. Returns Int on success.

Telegram documentation: https://core.telegram.org/bots/api#getchatmembercount

Parameters
chat_id –

Returns
API reply.

get_chat_members_count(***kwargs*)

get_chat_menu_button(*chat_id: Optional[Union[int, str]] = None*) → *MenuButton*

Use this method to get the current value of the bot's menu button in a private chat, or the default menu button. Returns MenuButton on success.

Telegram Documentation: https://core.telegram.org/bots/api#getchatmenubutton

Parameters
chat_id – Unique identifier for the target private chat. If not specified, default bot's menu button will be returned.

Returns
types.MenuButton

get_file(*file_id: str*) → *File*

Use this method to get basic info about a file and prepare it for downloading. For the moment, bots can download files of up to 20MB in size. On success, a File object is returned. It is guaranteed that the link will be valid for at least 1 hour. When the link expires, a new one can be requested by calling get_file again.

Telegram documentation: https://core.telegram.org/bots/api#getfile

Parameters
file_id – File identifier

get_file_url(*file_id: str*) → str

Get a valid URL for downloading a file.

Parameters
file_id – File identifier to get download URL for.

get_game_high_scores(*user_id: int*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*) → List[*GameHighScore*]

Gets top points and game play.

Telegram documentation: https://core.telegram.org/bots/api#getgamehighscores

Parameters

- **user_id** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –

Returns

get_me() → *User*

Returns basic information about the bot in form of a User object.

Telegram documentation: https://core.telegram.org/bots/api#getme

get_my_commands(*scope: Optional[*BotCommandScope*] = None*, *language_code: Optional[str] = None*) → List[*BotCommand*]

Use this method to get the current list of the bot's commands. Returns List of BotCommand on success.

Telegram documentation: https://core.telegram.org/bots/api#getmycommands

Parameters

- **scope** – The scope of users for which the commands are relevant. Defaults to BotCommandScopeDefault.
- **language_code** – A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

get_my_default_administrator_rights(*for_channels: Optional[bool] = None*) → *ChatAdministratorRights*

Use this method to get the current default administrator rights of the bot. Returns ChatAdministratorRights on success.

Telegram documentation: https://core.telegram.org/bots/api#getmydefaultadministratorrights

Parameters

for_channels – Pass True to get the default administrator rights of the bot in channels. Otherwise, the default administrator rights of the bot for groups and supergroups will be returned.

Returns

types.ChatAdministratorRights

get_state(*user_id: int*, *chat_id: Optional[int] = None*) → Optional[Union[int, str]]

Get current state of a user.

Parameters

- **user_id** –
- **chat_id** –

Returns

state of a user

get_sticker_set(*name: str*) → *StickerSet*

Use this method to get a sticker set. On success, a StickerSet object is returned.

Telegram documentation: https://core.telegram.org/bots/api#getstickerset

Parameters

name –

Returns

get_updates(*offset: Optional[int] = None*, *limit: Optional[int] = None*, *timeout: Optional[int] = 20*, *allowed_updates: Optional[List[str]] = None*, *long_polling_timeout: int = 20*) → List[*Update*]

Use this method to receive incoming updates using long polling (wiki). An Array of Update objects is returned.

Telegram documentation: https://core.telegram.org/bots/api#getupdates

Parameters

- **allowed_updates** – Array of string. List the types of updates you want your bot to receive.
- **offset** – Integer. Identifier of the first update to be returned.
- **limit** – Integer. Limits the number of updates to be retrieved.
- **timeout** – Integer. Request connection timeout
- **long_polling_timeout** – Timeout in seconds for long polling.

Returns
array of Updates

get_user_profile_photos(*user_id: int*, *offset: Optional[int] = None*, *limit: Optional[int] = None*) → *UserProfilePhotos*

Retrieves the user profile photos of the person with 'user_id'

Telegram documentation: https://core.telegram.org/bots/api#getuserprofilephotos

Parameters

- **user_id** – Integer - Unique identifier of the target user
- **offset** –
- **limit** –

Returns
API reply.

get_webhook_info(*timeout: Optional[int] = None*)

Use this method to get current webhook status. Requires no parameters. If the bot is using getUpdates, will return an object with the url field empty.

Telegram documentation: https://core.telegram.org/bots/api#getwebhookinfo

Parameters
timeout – Integer. Request connection timeout

Returns
On success, returns a WebhookInfo object.

infinity_polling(*timeout: int = 20*, *skip_pending: bool = False*, *long_polling_timeout: int = 20*, *logger_level=40*, *allowed_updates: Optional[List[str]] = None*, **args*, ***kwargs*)

Wrap polling with infinite loop and exception handling to avoid bot stops polling.

Parameters

- **timeout** – Request connection timeout
- **long_polling_timeout** – Timeout in seconds for long polling (see API docs)
- **skip_pending** – skip old updates
- **logger_level** – Custom (different from logger itself) logging level for infinity_polling logging. Use logger levels from logging as a value. None/NOTSET = no error logging
- **allowed_updates** – A list of the update types you want your bot to receive. For example, specify ["message", "edited_channel_post", "callback_query"] to only receive updates of these types. See util.update_types for a complete list of available update types. Specify an empty list to receive all update types except chat_member (default). If not specified, the previous setting will be used.

Please note that this parameter doesn't affect updates created before the call to the get_updates, so unwanted updates may be received for a short period of time.

inline_handler(*func*, ***kwargs*)

Inline call handler decorator

Parameters

- **func** –
- **kwargs** –

Returns

kick_chat_member(***kwargs*)

leave_chat(*chat_id: Union[int, str]*) → bool

Use this method for your bot to leave a group, supergroup or channel. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#leavechat

Parameters
chat_id –

Returns
API reply.

load_next_step_handlers(*filename='./.handler-saves/step.save'*, *del_file_after_loading=True*)

Load next step handlers from save file

This function is left to keep backward compatibility whose purpose was to load handlers from file with the help of FileHandlerBackend and is only recommended to use if next_step_backend was assigned as FileHandlerBackend before entering this function

Parameters

- **filename** – Filename of the file where handlers was saved
- **del_file_after_loading** – Is passed True, after loading save file will be deleted

load_reply_handlers(*filename='./.handler-saves/reply.save'*, *del_file_after_loading=True*)

Load reply handlers from save file

This function is left to keep backward compatibility whose purpose was to load handlers from file with the help of FileHandlerBackend and is only recommended to use if reply_backend was assigned as FileHandlerBackend before entering this function

Parameters

- **filename** – Filename of the file where handlers was saved
- **del_file_after_loading** – Is passed True, after loading save file will be deleted

log_out() → bool

Use this method to log out from the cloud Bot API server before launching the bot locally. You MUST log out the bot before running it locally, otherwise there is no guarantee that the bot will receive updates. After a successful call, you can immediately log in on a local server, but will not be able to log in back to the cloud Bot API server for 10 minutes. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#logout

message_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, *chat_types=None*, ***kwargs*)

Message handler decorator. This decorator can be used to decorate functions that must handle certain types of messages. All message handlers are tested in the order they were added.

Example:

```
bot = TeleBot('TOKEN')

# Handles all messages which text matches regexp.
@bot.message_handler(regexp='someregexp')
def command_help(message):
    bot.send_message(message.chat.id, 'Did someone call for help?')

# Handles messages in private chat
@bot.message_handler(chat_types=['private']) # You can add more chat types
def command_help(message):
    bot.send_message(message.chat.id, 'Private chat detected, sir!')

# Handle all sent documents of type 'text/plain'.
@bot.message_handler(func=lambda message: message.document.mime_type == 'text/
↪plain',
    content_types=['document'])
def command_handle_document(message):
    bot.send_message(message.chat.id, 'Document received, sir!')

# Handle all other messages.
@bot.message_handler(func=lambda message: True, content_types=['audio', 'photo',
↪ 'voice', 'video', 'document',
    'text', 'location', 'contact', 'sticker'])
def default_command(message):
    bot.send_message(message.chat.id, "This is the default command handler.")
```

Parameters

- **commands** – Optional list of strings (commands to handle).
- **regexp** – Optional regular expression.
- **func** – Optional lambda function. The lambda receives the message to test as the first parameter. It must return True if the command should handle the message.
- **content_types** – Supported message content types. Must be a list. Defaults to [‘text’].
- **chat_types** – list of chat types

middleware_handler(*update_types=None*)

Middleware handler decorator.

This decorator can be used to decorate functions that must be handled as middlewares before entering any other message handlers But, be careful and check type of the update inside the handler if more than one update_type is given

Example:

```
bot = TeleBot('TOKEN')

# Print post message text before entering to any post_channel handlers
@bot.middleware_handler(update_types=['channel_post', 'edited_channel_post'])
def print_channel_post_text(bot_instance, channel_post):
    print(channel_post.text)

# Print update id before entering to any handlers
@bot.middleware_handler()
def print_channel_post_text(bot_instance, update):
    print(update.update_id)
```

Parameters

update_types – Optional list of update types that can be passed into the middleware handler.

my_chat_member_handler(*func=None*, ***kwargs*)

my_chat_member handler.

Parameters

- **func** –
- **kwargs** –

Returns

pin_chat_message(*chat_id: Union[int, str]*, *message_id: int*, *disable_notification: Optional[bool] = False*) → bool

Use this method to pin a message in a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#pinchatmessage

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **message_id** – Int: Identifier of a message to pin
- **disable_notification** – Bool: Pass True, if it is not necessary to send a notification to all group members about the new pinned message

Returns

poll_answer_handler(*func=None*, ***kwargs*)

Poll_answer request handler

Parameters

- **func** –
- **kwargs** –

Returns

poll_handler(*func*, ***kwargs*)

Poll request handler

Parameters

- **func** –
- **kwargs** –

Returns

polling(*non_stop: bool = False*, *skip_pending=False*, *interval: int = 0*, *timeout: int = 20*, *long_polling_timeout: int = 20*, *logger_level=40*, *allowed_updates: Optional[List[str]] = None*, *none_stop: Optional[bool] = None*)

This function creates a new Thread that calls an internal __retrieve_updates function. This allows the bot to retrieve Updates automagically and notify listeners and message handlers accordingly.

Warning: Do not call this function more than once!

Always get updates.

Parameters

- **interval** – Delay between two update retrivals
- **non_stop** – Do not stop polling when an ApiException occurs.
- **timeout** – Request connection timeout
- **skip_pending** – skip old updates
- **long_polling_timeout** – Timeout in seconds for long polling (see API docs)
- **logger_level** – Custom (different from logger itself) logging level for infinity_polling logging. Use logger levels from logging as a value. None/NOTSET = no error logging
- **allowed_updates** – A list of the update types you want your bot to receive. For example, specify [“message”, “edited_channel_post”, “callback_query”] to only receive updates of these types. See util.update_types for a complete list of available update types. Specify an empty list to receive all update types except chat_member (default). If not specified, the previous setting will be used.

  Please note that this parameter doesn't affect updates created before the call to the get_updates, so unwanted updates may be received for a short period of time.
- **none_stop** – Deprecated, use non_stop. Old typo f***up compatibility

Returns

pre_checkout_query_handler(*func*, ***kwargs*)

Pre-checkout request handler

Parameters

- **func** –
- **kwargs** –

Returns

process_middlewares(*update*)

process_new_callback_query(*new_callback_querys*)

process_new_channel_posts(*channel_post*)

process_new_chat_join_request(*chat_join_request*)

process_new_chat_member(*chat_members*)

process_new_chosen_inline_query(*new_chosen_inline_querys*)

process_new_edited_channel_posts(*edited_channel_post*)

process_new_edited_messages(*edited_message*)

process_new_inline_query(*new_inline_querys*)

process_new_messages(*new_messages*)

process_new_my_chat_member(*my_chat_members*)

process_new_poll(*polls*)

process_new_poll_answer(*poll_answers*)

process_new_pre_checkout_query(*pre_checkout_querys*)

process_new_shipping_query(*new_shipping_querys*)

process_new_updates(*updates*)

Processes new updates. Just pass list of subclasses of Update to this method.

Parameters

updates – List of Update objects

promote_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *can_change_info: Optional[bool] = None*, *can_post_messages: Optional[bool] = None*, *can_edit_messages: Optional[bool] = None*, *can_delete_messages: Optional[bool] = None*, *can_invite_users: Optional[bool] = None*, *can_restrict_members: Optional[bool] = None*, *can_pin_messages: Optional[bool] = None*, *can_promote_members: Optional[bool] = None*, *is_anonymous: Optional[bool] = None*, *can_manage_chat: Optional[bool] = None*, *can_manage_video_chats: Optional[bool] = None*, *can_manage_voice_chats: Optional[bool] = None*) → bool

Use this method to promote or demote a user in a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Pass False for all boolean parameters to demote a user.

Telegram documentation: https://core.telegram.org/bots/api#promotechatmember

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **user_id** – Int : Unique identifier of the target user
- **can_change_info** – Bool: Pass True, if the administrator can change chat title, photo and other settings
- **can_post_messages** – Bool : Pass True, if the administrator can create channel posts, channels only
- **can_edit_messages** – Bool : Pass True, if the administrator can edit messages of other users, channels only
- **can_delete_messages** – Bool : Pass True, if the administrator can delete messages of other users
- **can_invite_users** – Bool : Pass True, if the administrator can invite new users to the chat

- **can_restrict_members** – Bool: Pass True, if the administrator can restrict, ban or unban chat members
- **can_pin_messages** – Bool: Pass True, if the administrator can pin messages, supergroups only
- **can_promote_members** – Bool: Pass True, if the administrator can add new administrators with a subset of his own privileges or demote administrators that he has promoted, directly or indirectly (promoted by administrators that were appointed by him)
- **is_anonymous** – Bool: Pass True, if the administrator's presence in the chat is hidden
- **can_manage_chat** – Bool: Pass True, if the administrator can access the chat event log, chat statistics, message statistics in channels, see channel members, see anonymous administrators in supergroups and ignore slow mode. Implied by any other administrator privilege
- **can_manage_video_chats** – Bool: Pass True, if the administrator can manage voice chats For now, bots can use this privilege only for passing to other administrators.
- **can_manage_voice_chats** – Deprecated, use can_manage_video_chats.

Returns
True on success.

register_callback_query_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers callback query handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_channel_post_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *pass_bot=False*, ***kwargs*)

Registers channel post message handler.

Parameters

- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands
- **regexp** –
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_chat_join_request_handler(*callback*, *func=None*, *pass_bot=False*, ***kwargs*)

Registers chat join request handler.

Parameters

- **callback** – function to be called

- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_chat_member_handler(*callback*, *func=None*, *pass_bot=False*, ***kwargs*)

Registers chat member handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_chosen_inline_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers chosen inline handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_edited_channel_post_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *pass_bot=False*, ***kwargs*)

Registers edited channel post message handler.

Parameters

- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands
- **regexp** –
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_edited_message_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *chat_types=None*, *pass_bot=False*, ***kwargs*)

Registers edited message handler.

Parameters

- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands

- **regexp** –
- **func** –
- **chat_types** – True for private chat
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_for_reply(*message: Message*, *callback: Callable*, **args*, ***kwargs*) → None

Registers a callback function to be notified when a reply to *message* arrives.

Warning: In case *callback* as lambda function, saving reply handlers will not work.

Parameters

- **message** – The message for which we are awaiting a reply.
- **callback** – The callback function to be called when a reply arrives. Must accept one *message* parameter, which will contain the replied message.

register_for_reply_by_message_id(*message_id: int*, *callback: Callable*, **args*, ***kwargs*) → None

Registers a callback function to be notified when a reply to *message* arrives.

Warning: In case *callback* as lambda function, saving reply handlers will not work.

Parameters

- **message_id** – The id of the message for which we are awaiting a reply.
- **callback** – The callback function to be called when a reply arrives. Must accept one *message* parameter, which will contain the replied message.

register_inline_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers inline handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_message_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *chat_types=None*, *pass_bot=False*, ***kwargs*)

Registers message handler.

Parameters

- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands
- **regexp** –
- **func** –
- **chat_types** – True for private chat

- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_middleware_handler(*callback*, *update_types=None*)

Middleware handler decorator.

This function will create a decorator that can be used to decorate functions that must be handled as middlewares before entering any other message handlers But, be careful and check type of the update inside the handler if more than one update_type is given

Example:

bot = TeleBot('TOKEN')

bot.register_middleware_handler(print_channel_post_text, update_types=['channel_post', 'edited_channel_post'])

Parameters

- **callback** –
- **update_types** – Optional list of update types that can be passed into the middleware handler.

register_my_chat_member_handler(*callback*, *func=None*, *pass_bot=False*, ***kwargs*)

Registers my chat member handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_next_step_handler(*message:* Message, *callback: Callable*, **args*, ***kwargs*) → None

Registers a callback function to be notified when new message arrives after *message*.

Warning: In case *callback* as lambda function, saving next step handlers will not work.

Parameters

- **message** – The message for which we want to handle new message in the same chat.
- **callback** – The callback function which next new message arrives.
- **args** – Args to pass in callback func
- **kwargs** – Args to pass in callback func

register_next_step_handler_by_chat_id(*chat_id: Union[int, str]*, *callback: Callable*, **args*, ***kwargs*) → None

Registers a callback function to be notified when new message arrives after *message*.

Warning: In case *callback* as lambda function, saving next step handlers will not work.

Parameters

- **chat_id** – The chat for which we want to handle new message.
- **callback** – The callback function which next new message arrives.

- **args** – Args to pass in callback func
- **kwargs** – Args to pass in callback func

register_poll_answer_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers poll answer handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_poll_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers poll handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_pre_checkout_query_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers pre-checkout request handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

register_shipping_query_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers shipping query handler.

Parameters

- **callback** – function to be called
- **func** –
- **pass_bot** – Pass TeleBot to handler.

Returns
decorated function

remove_webhook()

reply_to(*message:* Message, *text: str*, ***kwargs*) → *Message*

Convenience function for *send_message(message.chat.id, text, reply_to_message_id=message.message_id, **kwargs)*

Parameters

- **message** –
- **text** –
- **kwargs** –

Returns

reset_data(*user_id: int*, *chat_id: Optional[int] = None*)

Reset data for a user in chat.

Parameters

- **user_id** –
- **chat_id** –

restrict_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *until_date: Optional[Union[int, datetime]] = None*, *can_send_messages: Optional[bool] = None*, *can_send_media_messages: Optional[bool] = None*, *can_send_polls: Optional[bool] = None*, *can_send_other_messages: Optional[bool] = None*, *can_add_web_page_previews: Optional[bool] = None*, *can_change_info: Optional[bool] = None*, *can_invite_users: Optional[bool] = None*, *can_pin_messages: Optional[bool] = None*) → bool

Use this method to restrict a user in a supergroup. The bot must be an administrator in the supergroup for this to work and must have the appropriate admin rights. Pass True for all boolean parameters to lift restrictions from a user.

Telegram documentation: https://core.telegram.org/bots/api#restrictchatmember

Parameters

- **chat_id** – Int or String : Unique identifier for the target group or username of the target supergroup or channel (in the format @channelusername)
- **user_id** – Int : Unique identifier of the target user
- **until_date** – Date when restrictions will be lifted for the user, unix time. If user is restricted for more than 366 days or less than 30 seconds from the current time, they are considered to be restricted forever
- **can_send_messages** – Pass True, if the user can send text messages, contacts, locations and venues
- **can_send_media_messages** – Pass True, if the user can send audios, documents, photos, videos, video notes and voice notes, implies can_send_messages
- **can_send_polls** – Pass True, if the user is allowed to send polls, implies can_send_messages
- **can_send_other_messages** – Pass True, if the user can send animations, games, stickers and use inline bots, implies can_send_media_messages
- **can_add_web_page_previews** – Pass True, if the user may add web page previews to their messages, implies can_send_media_messages
- **can_change_info** – Pass True, if the user is allowed to change the chat title, photo and other settings. Ignored in public supergroups
- **can_invite_users** – Pass True, if the user is allowed to invite new users to the chat, implies can_invite_users
- **can_pin_messages** – Pass True, if the user is allowed to pin messages. Ignored in public supergroups

Returns
True on success

retrieve_data(*user_id: int*, *chat_id: Optional[int] = None*) → Optional[Union[int, str]]

revoke_chat_invite_link(*chat_id: Union[int, str]*, *invite_link: str*) → *ChatInviteLink*

Use this method to revoke an invite link created by the bot. Note: If the primary link is revoked, a new link is automatically generated The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#revokechatinvitelink

Parameters

- **chat_id** – Id: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **invite_link** – The invite link to revoke

Returns

send_animation(*chat_id: Union[int, str]*, *animation: Union[Any, str]*, *duration: Optional[int] = None*, *width: Optional[int] = None*, *height: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound).

Telegram documentation: https://core.telegram.org/bots/api#sendanimation

Parameters

- **chat_id** – Integer : Unique identifier for the message recipient — User or GroupChat id
- **animation** – InputFile or String : Animation to send. You can either pass a file_id as String to resend an animation that is already on the Telegram server
- **duration** – Integer : Duration of sent video in seconds
- **width** – Integer : Video width
- **height** – Integer : Video height
- **thumb** – InputFile or String : Thumbnail of the file sent
- **caption** – String : Animation caption (may also be used when resending animation by file_id).
- **parse_mode** –
- **protect_content** –
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** –
- **timeout** –
- **caption_entities** –

- **allow_sending_without_reply** –

Returns

send_audio(*chat_id: Union[int, str]*, *audio: Union[Any, str]*, *caption: Optional[str] = None*, *duration: Optional[int] = None*, *performer: Optional[str] = None*, *title: Optional[str] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *parse_mode: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .mp3 format.

Telegram documentation: https://core.telegram.org/bots/api#sendaudio

Parameters

- **chat_id** – Unique identifier for the message recipient
- **audio** – Audio file to send.
- **caption** –
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **reply_markup** –
- **parse_mode** –
- **disable_notification** –
- **timeout** –
- **thumb** –
- **caption_entities** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

Message

send_chat_action(*chat_id: Union[int, str]*, *action: str*, *timeout: Optional[int] = None*) → bool

Use this method when you need to tell the user that something is happening on the bot's side. The status is set for 5 seconds or less (when a message arrives from your bot, Telegram clients clear its typing status).

Telegram documentation: https://core.telegram.org/bots/api#sendchataction

Parameters

- **chat_id** –
- **action** – One of the following strings: 'typing', 'upload_photo', 'record_video', 'upload_video', 'record_audio', 'upload_audio', 'upload_document', 'find_location', 'record_video_note', 'upload_video_note'.

- **timeout** –

Returns
API reply. :type: boolean

send_contact(*chat_id: Union[int, str]*, *phone_number: str*, *first_name: str*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send phone contacts.

Telegram documentation: https://core.telegram.org/bots/api#sendcontact

Parameters

- **chat_id** – Integer or String : Unique identifier for the target chat or username of the target channel
- **phone_number** – String : Contact's phone number
- **first_name** – String : Contact's first name
- **last_name** – String : Contact's last name
- **vcard** – String : Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

send_dice(*chat_id: Union[int, str]*, *emoji: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send dices.

Telegram documentation: https://core.telegram.org/bots/api#senddice

Parameters

- **chat_id** –
- **emoji** –
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –

- **protect_content** –

Returns
Message

send_document(*chat_id: Union[int, str]*, *document: Union[Any, str]*, *reply_to_message_id: Optional[int] = None*, *caption: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *parse_mode: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *visible_file_name: Optional[str] = None*, *disable_content_type_detection: Optional[bool] = None*, *data: Optional[Union[Any, str]] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send general files.

Telegram documentation: https://core.telegram.org/bots/api#senddocument

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **document** – (document) File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **caption** – Document caption (may also be used when resending documents by file_id), 0-1024 characters after entities parsing
- **reply_markup** –
- **parse_mode** – Mode for parsing entities in the document caption
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **timeout** –
- **thumb** – InputFile or String : Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass "attach://<file_attach_name>" if the thumbnail was uploaded using multipart/form-data under <file_attach_name>
- **caption_entities** –
- **allow_sending_without_reply** –
- **visible_file_name** – allows to define file name that will be visible in the Telegram instead of original file name
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **data** – function typo miss compatibility: do not use it
- **protect_content** –

Returns
API reply.

send_game(*chat_id: Union[int, str]*, *game_short_name: str*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Used to send the game.

Telegram documentation: https://core.telegram.org/bots/api#sendgame

Parameters

- **chat_id** –
- **game_short_name** –
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

send_invoice(*chat_id: Union[int, str]*, *title: str*, *description: str*, *invoice_payload: str*, *provider_token: str*, *currency: str*, *prices: List[LabeledPrice]*, *start_parameter: Optional[str] = None*, *photo_url: Optional[str] = None*, *photo_size: Optional[int] = None*, *photo_width: Optional[int] = None*, *photo_height: Optional[int] = None*, *need_name: Optional[bool] = None*, *need_phone_number: Optional[bool] = None*, *need_email: Optional[bool] = None*, *need_shipping_address: Optional[bool] = None*, *send_phone_number_to_provider: Optional[bool] = None*, *send_email_to_provider: Optional[bool] = None*, *is_flexible: Optional[bool] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *provider_data: Optional[str] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *max_tip_amount: Optional[int] = None*, *suggested_tip_amounts: Optional[List[int]] = None*, *protect_content: Optional[bool] = None*) → *Message*

Sends invoice.

Telegram documentation: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **chat_id** – Unique identifier for the target private chat
- **title** – Product name
- **description** – Product description
- **invoice_payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use for your internal processes.
- **provider_token** – Payments provider token, obtained via @Botfather
- **currency** – Three-letter ISO 4217 currency code, see https://core.telegram.org/bots/payments#supported-currencies
- **prices** – Price breakdown, a list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.)

- **start_parameter** – Unique deep-linking parameter that can be used to generate this invoice when used as a start parameter
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass True, if you require the user's full name to complete the order
- **need_phone_number** – Pass True, if you require the user's phone number to complete the order
- **need_email** – Pass True, if you require the user's email to complete the order
- **need_shipping_address** – Pass True, if you require the user's shipping address to complete the order
- **is_flexible** – Pass True, if the final price depends on the shipping method
- **send_phone_number_to_provider** – Pass True, if user's phone number should be sent to provider
- **send_email_to_provider** – Pass True, if user's email address should be sent to provider
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay total price' button will be shown. If not empty, the first button must be a Pay button
- **provider_data** – A JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **timeout** –
- **allow_sending_without_reply** –
- **max_tip_amount** – The maximum accepted amount for tips in the smallest units of the currency
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the smallest units of the currency. At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed max_tip_amount.
- **protect_content** –

Returns

send_location(*chat_id: Union[int, str]*, *latitude: float*, *longitude: float*, *live_period: Optional[int] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send point on the map.

Telegram documentation: https://core.telegram.org/bots/api#sendlocation

Parameters

- **chat_id** –
- **latitude** –
- **longitude** –
- **live_period** –
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** –
- **timeout** –
- **horizontal_accuracy** –
- **heading** –
- **proximity_alert_radius** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

API reply.

send_media_group(*chat_id: Union[int, str]*, *media: List[Union[*InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo*]]*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*) → List[*Message*]

Send a group of photos or videos as an album. On success, an array of the sent Messages is returned.

Telegram documentation: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **chat_id** –
- **media** –
- **disable_notification** –
- **protect_content** –
- **reply_to_message_id** –
- **timeout** –
- **allow_sending_without_reply** –

Returns

send_message(*chat_id: Union[int, str]*, *text: str*, *parse_mode: Optional[str] = None*, *entities: Optional[List[MessageEntity]] = None*, *disable_web_page_preview: Optional[bool] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to send text messages.

Warning: Do not send more than about 4000 characters each message, otherwise you'll risk an HTTP 414 error. If you must send more than 4000 characters, use the *split_string* or *smart_split* function in util.py.

Telegram documentation: https://core.telegram.org/bots/api#sendmessage

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **text** – Text of the message to be sent
- **parse_mode** – Send Markdown or HTML, if you want Telegram apps to show bold, italic, fixed-width text or inline URLs in your bot's message.
- **entities** – List of special entities that appear in message text, which can be specified instead of parse_mode
- **disable_web_page_preview** – Disables link previews for links in this message
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** – If True, the message content will be hidden for all users except for the target user
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **allow_sending_without_reply** – Pass True, if the message should be sent even if the specified replied-to message is not found
- **reply_markup** – Additional interface options. A JSON-serialized object for an inline keyboard, custom reply keyboard, instructions to remove reply keyboard or to force a reply from the user.
- **timeout** –

Returns

API reply.

send_photo(*chat_id: Union[int, str]*, *photo: Union[Any, str]*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[MessageEntity]] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to send photos. On success, the sent Message is returned.

Telegram documentation: https://core.telegram.org/bots/api#sendphoto

Parameters

- **chat_id** –
- **photo** –

- **caption** –
- **parse_mode** –
- **caption_entities** –
- **disable_notification** –
- **protect_content** –
- **reply_to_message_id** –
- **allow_sending_without_reply** –
- **reply_markup** –
- **timeout** –

Returns
Message

send_poll(*chat_id: Union[int, str]*, *question: str*, *options: List[str]*, *is_anonymous: Optional[bool] = None*, *type: Optional[str] = None*, *allows_multiple_answers: Optional[bool] = None*, *correct_option_id: Optional[int] = None*, *explanation: Optional[str] = None*, *explanation_parse_mode: Optional[str] = None*, *open_period: Optional[int] = None*, *close_date: Optional[Union[int, datetime]] = None*, *is_closed: Optional[bool] = None*, *disable_notification: Optional[bool] = False*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *timeout: Optional[int] = None*, *explanation_entities: Optional[List[*MessageEntity*]] = None*, *protect_content: Optional[bool] = None*) → *Message*

Sends a poll.

Telegram documentation: https://core.telegram.org/bots/api#sendpoll

Parameters

- **chat_id** –
- **question** –
- **options** – array of str with answers
- **is_anonymous** –
- **type** –
- **allows_multiple_answers** –
- **correct_option_id** –
- **explanation** –
- **explanation_parse_mode** –
- **open_period** –
- **close_date** –
- **is_closed** –
- **disable_notification** –
- **reply_to_message_id** –
- **allow_sending_without_reply** –
- **reply_markup** –

- **timeout** –
- **explanation_entities** –
- **protect_content** –

Returns

send_sticker(*chat_id: Union[int, str]*, *sticker: Union[Any, str]*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *data: Optional[Union[Any, str]] = None*) → *Message*

Use this method to send .webp stickers.

Telegram documentation: https://core.telegram.org/bots/api#sendsticker

Parameters

- **chat_id** –
- **sticker** –
- **data** –
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** – to disable the notification
- **timeout** – timeout
- **allow_sending_without_reply** –
- **protect_content** –
- **data** – function typo miss compatibility: do not use it

Returns

API reply.

send_venue(*chat_id: Union[int, str]*, *latitude: float*, *longitude: float*, *title: str*, *address: str*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send information about a venue.

Telegram documentation: https://core.telegram.org/bots/api#sendvenue

Parameters

- **chat_id** – Integer or String : Unique identifier for the target chat or username of the target channel
- **latitude** – Float : Latitude of the venue
- **longitude** – Float : Longitude of the venue
- **title** – String : Name of the venue
- **address** – String : Address of the venue

- **foursquare_id** – String : Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, "arts_entertainment/default", "arts_entertainment/aquarium" or "food/icecream".)
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –
- **google_place_id** –
- **google_place_type** –
- **protect_content** –

Returns

send_video(*chat_id: Union[int, str]*, *video: Union[Any, str]*, *duration: Optional[int] = None*, *width: Optional[int] = None*, *height: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *supports_streaming: Optional[bool] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*, *data: Optional[Union[Any, str]] = None*) → *Message*

Use this method to send video files, Telegram clients support mp4 videos (other formats may be sent as Document).

Telegram documentation: https://core.telegram.org/bots/api#sendvideo

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **video** – Video to send. You can either pass a file_id as String to resend a video that is already on the Telegram servers, or upload a new video file using multipart/form-data.
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumb** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass "attach://<file_attach_name>" if the thumbnail was uploaded using multipart/form-data under <file_attach_name>.
- **caption** – Video caption (may also be used when resending videos by file_id), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption
- **caption_entities** –

- **supports_streaming** – Pass True, if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** –
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **allow_sending_without_reply** –
- **reply_markup** –
- **timeout** –
- **data** – function typo miss compatibility: do not use it

send_video_note(*chat_id: Union[int, str]*, *data: Union[Any, str]*, *duration: Optional[int] = None*, *length: Optional[int] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

As of v.4.0, Telegram clients support rounded square mp4 videos of up to 1 minute long. Use this method to send
video messages.

Telegram documentation: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **chat_id** – Integer : Unique identifier for the message recipient — User or GroupChat id
- **data** – InputFile or String : Video note to send. You can either pass a file_id as String to resend a video that is already on the Telegram server
- **duration** – Integer : Duration of sent video in seconds
- **length** – Integer : Video width and height, Can't be None and should be in range of (0, 640)
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** –
- **timeout** –
- **thumb** – InputFile or String : Thumbnail of the file sent
- **allow_sending_without_reply** –
- **protect_content** –

Returns

send_voice(*chat_id: Union[int, str]*, *voice: Union[Any, str]*, *caption: Optional[str] = None*, *duration: Optional[int] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *parse_mode: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message.

Telegram documentation: https://core.telegram.org/bots/api#sendvoice

Parameters

- **chat_id** – Unique identifier for the message recipient.
- **voice** –
- **caption** –
- **duration** – Duration of sent audio in seconds
- **reply_to_message_id** –
- **reply_markup** –
- **parse_mode** –
- **disable_notification** –
- **timeout** –
- **caption_entities** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

Message

set_chat_administrator_custom_title(*chat_id: Union[int, str]*, *user_id: int*, *custom_title: str*) → bool

Use this method to set a custom title for an administrator in a supergroup promoted by the bot.

Telegram documentation: https://core.telegram.org/bots/api#setchatadministratorcustomtitle

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **user_id** – Unique identifier of the target user
- **custom_title** – New custom title for the administrator; 0-16 characters, emoji are not allowed

Returns

True on success.

set_chat_description(*chat_id: Union[int, str]*, *description: Optional[str] = None*) → bool

Use this method to change the description of a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#setchatdescription

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **description** – Str: New chat description, 0-255 characters

Returns
True on success.

set_chat_menu_button(*chat_id: Optional[Union[int, str]] = None*, *menu_button: Optional[MenuButton] = None*) → bool

Use this method to change the bot's menu button in a private chat, or the default menu button. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setchatmenubutton

Parameters

- **chat_id** – Unique identifier for the target private chat. If not specified, default bot's menu button will be changed.
- **menu_button** – A JSON-serialized object for the new bot's menu button. Defaults to MenuButtonDefault

set_chat_permissions(*chat_id: Union[int, str]*, *permissions:* ChatPermissions) → bool

Use this method to set default chat permissions for all members. The bot must be an administrator in the group or a supergroup for this to work and must have the can_restrict_members admin rights.

Telegram documentation: https://core.telegram.org/bots/api#setchatpermissions

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **permissions** – New default chat permissions

Returns
True on success

set_chat_photo(*chat_id: Union[int, str]*, *photo: Any*) → bool

Use this method to set a new profile photo for the chat. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success. Note: In regular groups (non-supergroups), this method will only work if the 'All Members Are Admins' setting is off in the target group.

Telegram documentation: https://core.telegram.org/bots/api#setchatphoto

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **photo** – InputFile: New chat photo, uploaded using multipart/form-data

Returns

set_chat_sticker_set(*chat_id: Union[int, str]*, *sticker_set_name: str*) → *StickerSet*

Use this method to set a new group sticker set for a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Use the field can_set_sticker_set optionally returned in getChat requests to check if the bot can use this method. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setchatstickerset

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **sticker_set_name** – Name of the sticker set to be set as the group sticker set

Returns
API reply.

`set_chat_title`(*chat_id: Union[int, str]*, *title: str*) → bool

Use this method to change the title of a chat. Titles can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success. Note: In regular groups (non-supergroups), this method will only work if the 'All Members Are Admins' setting is off in the target group.

Telegram documentation: https://core.telegram.org/bots/api#setchattitle

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **title** – New chat title, 1-255 characters

Returns

`set_game_score`(*user_id: Union[int, str]*, *score: int*, *force: Optional[bool] = None*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *disable_edit_message: Optional[bool] = None*) → Union[*Message*, bool]

Sets the value of points in the game to a specific user.

Telegram documentation: https://core.telegram.org/bots/api#setgamecore

Parameters

- **user_id** –
- **score** –
- **force** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **disable_edit_message** –

Returns

`set_my_commands`(*commands: List[BotCommand]*, *scope: Optional[BotCommandScope] = None*, *language_code: Optional[str] = None*) → bool

Use this method to change the list of the bot's commands.

Telegram documentation: https://core.telegram.org/bots/api#setmycommands

Parameters

- **commands** – List of BotCommand. At most 100 commands can be specified.
- **scope** – The scope of users for which the commands are relevant. Defaults to BotCommandScopeDefault.
- **language_code** – A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

Returns

set_my_default_administrator_rights(*rights: Optional[ChatAdministratorRights] = None*, *for_channels: Optional[bool] = None*) → bool

Use this method to change the default administrator rights requested by the bot when it's added as an administrator to groups or channels. These rights will be suggested to users, but they are are free to modify the list before adding the bot. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setmydefaultadministratorrights

Parameters

- **rights** – A JSON-serialized object describing new default administrator rights. If not specified, the default administrator rights will be cleared.
- **for_channels** – Pass True to change the default administrator rights of the bot in channels. Otherwise, the default administrator rights of the bot for groups and supergroups will be changed.

set_state(*user_id: int*, *state: Union[int, str, State]*, *chat_id: Optional[int] = None*) → None

Sets a new state of a user.

Parameters

- **user_id** –
- **state** – new state. can be string or integer.
- **chat_id** –

set_sticker_position_in_set(*sticker: str*, *position: int*) → bool

Use this method to move a sticker in a set created by the bot to a specific position . Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setstickerpositioninset

Parameters

- **sticker** –
- **position** –

Returns

set_sticker_set_thumb(*name: str*, *user_id: int*, *thumb: Optional[Union[Any, str]] = None*)

Use this method to set the thumbnail of a sticker set. Animated thumbnails can be set for animated sticker sets only. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setstickersetthumb

Parameters

- **name** – Sticker set name
- **user_id** – User identifier
- **thumb** –

set_update_listener(*listener*)

set_webhook(*url=None*, *certificate=None*, *max_connections=None*, *allowed_updates=None*, *ip_address=None*, *drop_pending_updates=None*, *timeout=None*, *secret_token=None*)

Use this method to specify a url and receive incoming updates via an outgoing webhook. Whenever there is an update for the bot, we will send an HTTPS POST request to the specified url, containing a JSON-serialized Update. In case of an unsuccessful request, we will give up after a reasonable amount of attempts. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setwebhook

Parameters

- **url** – HTTPS url to send updates to. Use an empty string to remove webhook integration
- **certificate** – Upload your public key certificate so that the root certificate in use can be checked. See our self-signed guide for details.
- **max_connections** – Maximum allowed number of simultaneous HTTPS connections to the webhook for update delivery, 1-100. Defaults to 40. Use lower values to limit the load on your bot's server, and higher values to increase your bot's throughput.
- **allowed_updates** – A JSON-serialized list of the update types you want your bot to receive. For example, specify ["message", "edited_channel_post", "callback_query"] to only receive updates of these types. See Update for a complete list of available update types. Specify an empty list to receive all updates regardless of type (default). If not specified, the previous setting will be used.
- **ip_address** – The fixed IP address which will be used to send webhook requests instead of the IP address resolved through DNS
- **drop_pending_updates** – Pass True to drop all pending updates
- **timeout** – Integer. Request connection timeout
- **secret_token** – Secret token to be used to verify the webhook request.

Returns
API reply.

setup_middleware(*middleware:* BaseMiddleware)

Register middleware

Parameters
middleware – Subclass of *telebot.handler_backends.BaseMiddleware*

Returns
None

shipping_query_handler(*func*, ***kwargs*)

Shipping request handler

Parameters

- **func** –
- **kwargs** –

Returns

stop_bot()

stop_message_live_location(*chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to stop updating a live location message sent by the bot or via the bot (for inline bots) before live_period expires

Telegram documentation: https://core.telegram.org/bots/api#stopmessagelivelocation

Parameters

- **chat_id** –

- **message_id** –
- **inline_message_id** –
- **reply_markup** –
- **timeout** –

Returns

stop_poll(*chat_id: Union[int, str]*, *message_id: int*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*) → *Poll*

Stops a poll.

Telegram documentation: https://core.telegram.org/bots/api#stoppoll

Parameters

- **chat_id** –
- **message_id** –
- **reply_markup** –

Returns

stop_polling()

unban_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *only_if_banned: Optional[bool] = False*) → bool

Use this method to unban a previously kicked user in a supergroup or channel. The user will not return to the group or channel automatically, but will be able to join via link, etc. The bot must be an administrator for this to work. By default, this method guarantees that after the call the user is not a member of the chat, but will be able to join it. So if the user is a member of the chat they will also be removed from the chat. If you don't want this, use the parameter only_if_banned.

Telegram documentation: https://core.telegram.org/bots/api#unbanchatmember

Parameters

- **chat_id** – Unique identifier for the target group or username of the target supergroup or channel (in the format @username)
- **user_id** – Unique identifier of the target user
- **only_if_banned** – Do nothing if the user is not banned

Returns
True on success

unban_chat_sender_chat(*chat_id: Union[int, str]*, *sender_chat_id: Union[int, str]*) → bool

Use this method to unban a previously banned channel chat in a supergroup or channel. The bot must be an administrator for this to work and must have the appropriate administrator rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#unbanchatsenderchat

Params
Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **sender_chat_id** – Unique identifier of the target sender chat

Returns
True on success.

unpin_all_chat_messages(*chat_id: Union[int, str]*) → bool

Use this method to unpin a all pinned messages in a supergroup chat. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#unpinallchatmessages

Parameters
chat_id – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)

Returns

unpin_chat_message(*chat_id: Union[int, str]*, *message_id: Optional[int] = None*) → bool

Use this method to unpin specific pinned message in a supergroup chat. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#unpinchatmessage

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **message_id** – Int: Identifier of a message to unpin

Returns

upload_sticker_file(*user_id: int*, *png_sticker: Union[Any, str]*) → *File*

Use this method to upload a .png file with a sticker for later use in createNewStickerSet and addStickerToSet methods (can be used multiple times). Returns the uploaded File on success.

Telegram documentation: https://core.telegram.org/bots/api#uploadstickerfile

Parameters

- **user_id** –
- **png_sticker** –

Returns

property user: *User*

The User object representing this bot. Equivalent to bot.get_me() but the result is cached so only one API call is needed

custom_filters file

class telebot.custom_filters.**AdvancedCustomFilter**

Bases: ABC

Simple Custom Filter base class. Create child class with check() method. Accepts two parameters, returns bool: True - filter passed, False - filter failed. message: Message class text: Filter value given in handler

Child classes should have .key property.

check(*message*, *text*)

Perform a check.

key: str = None

class telebot.custom_filters.**ChatFilter**

Bases: *AdvancedCustomFilter*

Check whether chat_id corresponds to given chat_id.

Example: @bot.message_handler(chat_id=[99999])

check(*message*, *text*)

Perform a check.

key: str = 'chat_id'

class telebot.custom_filters.**ForwardFilter**

Bases: *SimpleCustomFilter*

Check whether message was forwarded from channel or group.

Example:

@bot.message_handler(is_forwarded=True)

check(*message*)

Perform a check.

key: str = 'is_forwarded'

class telebot.custom_filters.**IsAdminFilter**(*bot*)

Bases: *SimpleCustomFilter*

Check whether the user is administrator / owner of the chat.

Example: @bot.message_handler(chat_types=['supergroup'], is_chat_admin=True)

check(*message*)

Perform a check.

key: str = 'is_chat_admin'

class telebot.custom_filters.**IsDigitFilter**

Bases: *SimpleCustomFilter*

Filter to check whether the string is made up of only digits.

Example: @bot.message_handler(is_digit=True)

check(*message*)

Perform a check.

key: str = 'is_digit'

class telebot.custom_filters.**IsReplyFilter**

Bases: *SimpleCustomFilter*

Check whether message is a reply.

Example:

@bot.message_handler(is_reply=True)

check(*message*)

Perform a check.

key: str = 'is_reply'

class telebot.custom_filters.**LanguageFilter**

Bases: *AdvancedCustomFilter*

Check users language_code.

Example:

@bot.message_handler(language_code=['ru'])

check(*message*, *text*)

Perform a check.

key: str = 'language_code'

class telebot.custom_filters.**SimpleCustomFilter**

Bases: ABC

Simple Custom Filter base class. Create child class with check() method. Accepts only message, returns bool value, that is compared with given in handler.

Child classes should have .key property.

check(*message*)

Perform a check.

key: str = None

class telebot.custom_filters.**StateFilter**(*bot*)

Bases: *AdvancedCustomFilter*

Filter to check state.

Example: @bot.message_handler(state=1)

check(*message*, *text*)

Perform a check.

key: str = 'state'

class telebot.custom_filters.**TextContainsFilter**

Bases: *AdvancedCustomFilter*

Filter to check Text message. key: text

Example: # Will respond if any message.text contains word 'account' @bot.message_handler(text_contains=['account'])

check(*message*, *text*)

Perform a check.

key: str = 'text_contains'

class telebot.custom_filters.**TextFilter**(*equals: Optional[str] = None*, *contains: Optional[Union[list, tuple]] = None*, *starts_with: Optional[Union[str, list, tuple]] = None*, *ends_with: Optional[Union[str, list, tuple]] = None*, *ignore_case: bool = False*)

Bases: object

Advanced text filter to check (types.Message, types.CallbackQuery, types.InlineQuery, types.Poll)

example of usage is in examples/custom_filters/advanced_text_filter.py

check(*obj: Union[*Message, CallbackQuery, InlineQuery, Poll*]*)

class telebot.custom_filters.**TextMatchFilter**

Bases: *AdvancedCustomFilter*

Filter to check Text message. key: text

Example: @bot.message_handler(text=['account'])

check(*message*, *text*)

Perform a check.

key: str = 'text'

class telebot.custom_filters.**TextStartsFilter**

Bases: *AdvancedCustomFilter*

Filter to check whether message starts with some text.

Example: # Will work if message.text starts with 'Sir'. @bot.message_handler(text_startswith='Sir')

check(*message*, *text*)

Perform a check.

key: str = 'text_startswith'

Synchronous storage for states

class telebot.storage.**StateContext**(*obj*, *chat_id*, *user_id*)

Bases: object

Class for data.

class telebot.storage.**StateMemoryStorage**

Bases: *StateStorageBase*

delete_state(*chat_id*, *user_id*)

Delete state for a particular user.

get_data(*chat_id*, *user_id*)

Get data for a user in a particular chat.

get_interactive_data(*chat_id*, *user_id*)

get_state(*chat_id*, *user_id*)

reset_data(*chat_id*, *user_id*)

Reset data for a particular user in a chat.

save(*chat_id*, *user_id*, *data*)

set_data(*chat_id*, *user_id*, *key*, *value*)

Set data for a user in a particular chat.

set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user.

! Note that you should create a record if it does not exist, and if a record with state already exists, you need to update a record.

class telebot.storage.**StatePickleStorage**(*file_path='./.state-save/states.pkl'*)

Bases: *StateStorageBase*

convert_old_to_new()

Use this function to convert old storage to new storage. This function is for people who was using pickle storage that was in version <=4.3.1.

create_dir()

Create directory .save-handlers.

delete_state(*chat_id*, *user_id*)

Delete state for a particular user.

get_data(*chat_id*, *user_id*)

Get data for a user in a particular chat.

get_interactive_data(*chat_id*, *user_id*)

get_state(*chat_id*, *user_id*)

read()

reset_data(*chat_id*, *user_id*)

Reset data for a particular user in a chat.

save(*chat_id*, *user_id*, *data*)

set_data(*chat_id*, *user_id*, *key*, *value*)

Set data for a user in a particular chat.

set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user.

! Note that you should create a record if it does not exist, and if a record with state already exists, you need to update a record.

update_data()

class telebot.storage.**StateRedisStorage**(*host='localhost'*, *port=6379*, *db=0*, *password=None*, *prefix='telebot_'*)

Bases: *StateStorageBase*

This class is for Redis storage. This will work only for states. To use it, just pass this class to: TeleBot(storage=StateRedisStorage())

delete_record(*key*)

Function to delete record from database. It has nothing to do with states. Made for backward compatibility

delete_state(*chat_id*, *user_id*)

Delete state for a particular user in a chat.

get_data(*chat_id*, *user_id*)

Get data of particular user in a particular chat.

get_interactive_data(*chat_id*, *user_id*)

Get Data in interactive way. You can use with() with this function.

get_record(*key*)

Function to get record from database. It has nothing to do with states. Made for backward compatibility

get_state(*chat_id*, *user_id*)

Get state of a user in a chat.

get_value(*chat_id*, *user_id*, *key*)

Get value for a data of a user in a chat.

reset_data(*chat_id*, *user_id*)

Reset data of a user in a chat.

save(*chat_id*, *user_id*, *data*)

set_data(*chat_id*, *user_id*, *key*, *value*)

Set data without interactive data.

set_record(*key*, *value*)

Function to set record to database. It has nothing to do with states. Made for backward compatibility

set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user in a chat.

class telebot.storage.**StateStorageBase**

Bases: object

delete_state(*chat_id*, *user_id*)

Delete state for a particular user.

get_data(*chat_id*, *user_id*)

Get data for a user in a particular chat.

get_state(*chat_id*, *user_id*)

reset_data(*chat_id*, *user_id*)

Reset data for a particular user in a chat.

save(*chat_id*, *user_id*, *data*)

set_data(*chat_id*, *user_id*, *key*, *value*)

Set data for a user in a particular chat.

set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user.

! Note that you should create a record if it does not exist, and if a record with state already exists, you need to update a record.

handler_backends file

class telebot.handler_backends.**BaseMiddleware**

Bases: object

Base class for middleware. Your middlewares should be inherited from this class.

post_process(*message*, *data*, *exception*)

pre_process(*message*, *data*)

class telebot.handler_backends.**CancelUpdate**

Bases: object

Class for canceling updates. Just return instance of this class in middleware to skip update. Update will skip handler and execution of post_process in middlewares.

class telebot.handler_backends.**FileHandlerBackend**(*handlers=None*, *filename='./.handler-saves/handlers.save'*, *delay=120*)

Bases: *HandlerBackend*

clear_handlers(*handler_group_id*)

static dump_handlers(*handlers*, *filename*, *file_mode='wb'*)

get_handlers(*handler_group_id*)

load_handlers(*filename=None*, *del_file_after_loading=True*)

register_handler(*handler_group_id*, *handler*)

static return_load_handlers(*filename*, *del_file_after_loading=True*)

save_handlers()

start_save_timer()

class telebot.handler_backends.**HandlerBackend**(*handlers=None*)

Bases: object

Class for saving (next step|reply) handlers

clear_handlers(*handler_group_id*)

get_handlers(*handler_group_id*)

register_handler(*handler_group_id*, *handler*)

class telebot.handler_backends.**MemoryHandlerBackend**(*handlers=None*)

Bases: *HandlerBackend*

clear_handlers(*handler_group_id*)

get_handlers(*handler_group_id*)

load_handlers(*filename*, *del_file_after_loading*)

register_handler(*handler_group_id*, *handler*)

class telebot.handler_backends.**RedisHandlerBackend**(*handlers=None*, *host='localhost'*, *port=6379*, *db=0*, *prefix='telebot'*, *password=None*)

Bases: *HandlerBackend*

clear_handlers(*handler_group_id*)

get_handlers(*handler_group_id*)

register_handler(*handler_group_id*, *handler*)

class telebot.handler_backends.**SkipHandler**

Bases: object

Class for skipping handlers. Just return instance of this class in middleware to skip handler. Update will go to post_process, but will skip execution of handler.

class telebot.handler_backends.**State**

Bases: object

class telebot.handler_backends.**StatesGroup**

Bases: object

1.3.5 AsyncTeleBot

AsyncTeleBot methods

class telebot.async_telebot.**AsyncTeleBot**(*token: str*, *parse_mode: ~typing.Optional[str] = None*, *offset=None*, *exception_handler=None*, *state_storage=<telebot.asyncio_storage.memory_storage.StateMemoryStorage object>*)

Bases: object

This is the main asynchronous class for Bot.

It allows you to add handlers for different kind of updates.

Usage:

```
from telebot.async_telebot import AsyncTeleBot
bot = AsyncTeleBot('token') # get token from @BotFather
```

See more examples in examples/ directory: https://github.com/eternnoir/pyTelegramBotAPI/tree/master/examples

add_callback_query_handler(*handler_dict*)

Adds a callback request handler. Note that you should use register_callback_query_handler to add callback_query_handler.

Parameters
handler_dict –

Returns

add_channel_post_handler(*handler_dict*)

Adds channel post handler. Note that you should use register_channel_post_handler to add channel_post_handler.

Parameters
handler_dict –

Returns

add_chat_join_request_handler(*handler_dict*)

Adds a chat_join_request handler. Note that you should use register_chat_join_request_handler to add chat_join_request_handler.

Parameters
handler_dict –

Returns

`add_chat_member_handler`(*handler_dict*)

Adds a chat_member handler. Note that you should use register_chat_member_handler to add chat_member_handler.

Parameters
`handler_dict` –

Returns

`add_chosen_inline_handler`(*handler_dict*)

Description: TBD Note that you should use register_chosen_inline_handler to add chosen_inline_handler.

Parameters
`handler_dict` –

Returns

`add_custom_filter`(*custom_filter*)

Create custom filter.

custom_filter: Class with check(message) method.

`async add_data`(*user_id: int*, *chat_id: Optional[int] = None*, ***kwargs*)

Add data to states.

Parameters

- `user_id` –
- `chat_id` –

`add_edited_channel_post_handler`(*handler_dict*)

Adds the edit channel post handler. Note that you should use register_edited_channel_post_handler to add edited_channel_post_handler.

Parameters
`handler_dict` –

Returns

`add_edited_message_handler`(*handler_dict*)

Adds the edit message handler. Note that you should use register_edited_message_handler to add edited_message_handler.

Parameters
`handler_dict` –

Returns

`add_inline_handler`(*handler_dict*)

Adds inline call handler. Note that you should use register_inline_handler to add inline_handler.

Parameters
`handler_dict` –

Returns

`add_message_handler`(*handler_dict*)

Adds a message handler. Note that you should use register_message_handler to add message_handler.

Parameters
`handler_dict` –

Returns

add_my_chat_member_handler(*handler_dict*)

Adds a my_chat_member handler. Note that you should use register_my_chat_member_handler to add my_chat_member_handler.

Parameters
handler_dict –

Returns

add_poll_answer_handler(*handler_dict*)

Adds a poll_answer request handler. Note that you should use register_poll_answer_handler to add poll_answer_handler.

Parameters
handler_dict –

Returns

add_poll_handler(*handler_dict*)

Adds a poll request handler. Note that you should use register_poll_handler to add poll_handler.

Parameters
handler_dict –

Returns

add_pre_checkout_query_handler(*handler_dict*)

Adds a pre-checkout request handler. Note that you should use register_pre_checkout_query_handler to add pre_checkout_query_handler.

Parameters
handler_dict –

Returns

add_shipping_query_handler(*handler_dict*)

Adds a shipping request handler. Note that you should use register_shipping_query_handler to add shipping_query_handler.

Parameters
handler_dict –

Returns

async add_sticker_to_set(*user_id: int*, *name: str*, *emojis: str*, *png_sticker: Optional[Union[Any, str]] = None*, *tgs_sticker: Optional[Union[Any, str]] = None*, *webm_sticker: Optional[Union[Any, str]] = None*, *mask_position: Optional[*MaskPosition*] = None*) → bool

Use this method to add a new sticker to a set created by the bot. It's required to pass *png_sticker* or *tgs_sticker*. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#addstickertoset

Parameters

- **user_id** –
- **name** –
- **emojis** –

- **png_sticker** – Required if *tgs_sticker* is None
- **tgs_sticker** – Required if *png_sticker* is None
- **mask_position** –

Webm_sticker
Returns

async **answer_callback_query**(*callback_query_id: int*, *text: Optional[str] = None*, *show_alert: Optional[bool] = None*, *url: Optional[str] = None*, *cache_time: Optional[int] = None*) → bool

Use this method to send answers to callback queries sent from inline keyboards. The answer will be displayed to the user as a notification at the top of the chat screen or as an alert.

Telegram documentation: https://core.telegram.org/bots/api#answercallbackquery

Parameters

- **callback_query_id** –
- **text** –
- **show_alert** –
- **url** –
- **cache_time** –

Returns

async **answer_inline_query**(*inline_query_id: str*, *results: List[Any]*, *cache_time: Optional[int] = None*, *is_personal: Optional[bool] = None*, *next_offset: Optional[str] = None*, *switch_pm_text: Optional[str] = None*, *switch_pm_parameter: Optional[str] = None*) → bool

Use this method to send answers to an inline query. On success, True is returned. No more than 50 results per query are allowed.

Telegram documentation: https://core.telegram.org/bots/api#answerinlinequery

Parameters

- **inline_query_id** – Unique identifier for the answered query
- **results** – Array of results for the inline query
- **cache_time** – The maximum amount of time in seconds that the result of the inline query may be cached on the server.
- **is_personal** – Pass True, if results may be cached on the server side only for the user that sent the query.
- **next_offset** – Pass the offset that a client should send in the next query with the same text to receive more results.
- **switch_pm_parameter** – If passed, clients will display a button with specified text that switches the user to a private chat with the bot and sends the bot a start message with the parameter switch_pm_parameter
- **switch_pm_text** – Parameter for the start message sent to the bot when user presses the switch button

Returns
True means success.

async answer_pre_checkout_query(*pre_checkout_query_id: int*, *ok: bool*, *error_message: Optional[str] = None*) → bool

Response to a request for pre-inspection.

Telegram documentation: https://core.telegram.org/bots/api#answerprecheckoutquery

Parameters

- **pre_checkout_query_id** –
- **ok** –
- **error_message** –

Returns

async answer_shipping_query(*shipping_query_id: str*, *ok: bool*, *shipping_options: Optional[List[ShippingOption]] = None*, *error_message: Optional[str] = None*) → bool

Asks for an answer to a shipping question.

Telegram documentation: https://core.telegram.org/bots/api#answershippingquery

Parameters

- **shipping_query_id** –
- **ok** –
- **shipping_options** –
- **error_message** –

Returns

async answer_web_app_query(*web_app_query_id: str*, *result:* InlineQueryResultBase) → *SentWebAppMessage*

Use this method to set the result of an interaction with a Web App and send a corresponding message on behalf of the user to the chat from which the query originated. On success, a SentWebAppMessage object is returned.

Telegram Documentation: https://core.telegram.org/bots/api#answerwebappquery

Parameters

- **web_app_query_id** – Unique identifier for the query to be answered
- **result** – A JSON-serialized object describing the message to be sent

Returns

async approve_chat_join_request(*chat_id: Union[str, int]*, *user_id: Union[int, str]*) → bool

Use this method to approve a chat join request. The bot must be an administrator in the chat for this to work and must have the can_invite_users administrator right. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#approvechatjoinrequest

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **user_id** – Unique identifier of the target user

Returns

True on success.

async ban_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *until_date: Optional[Union[int, datetime]] = None*, *revoke_messages: Optional[bool] = None*) → bool

Use this method to ban a user in a group, a supergroup or a channel. In the case of supergroups and channels, the user will not be able to return to the chat on their own using invite links, etc., unless unbanned first. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#banchatmember

Parameters

- **chat_id** – Int or string : Unique identifier for the target group or username of the target supergroup
- **user_id** – Int : Unique identifier of the target user
- **until_date** – Date when the user will be unbanned, unix time. If user is banned for more than 366 days or less than 30 seconds from the current time they are considered to be banned forever
- **revoke_messages** – Bool: Pass True to delete all messages from the chat for the user that is being removed. If False, the user will be able to see messages in the group that were sent before the user was removed. Always True for supergroups and channels.

Returns

boolean

async ban_chat_sender_chat(*chat_id: Union[int, str]*, *sender_chat_id: Union[int, str]*) → bool

Use this method to ban a channel chat in a supergroup or a channel. The owner of the chat will not be able to send messages and join live streams on behalf of the chat, unless it is unbanned first. The bot must be an administrator in the supergroup or channel for this to work and must have the appropriate administrator rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#banchatsenderchat

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **sender_chat_id** – Unique identifier of the target sender chat

Returns

True on success.

callback_query_handler(*func*, ***kwargs*)

Callback request handler decorator.

Parameters

- **func** –
- **kwargs** –

Returns

channel_post_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, ***kwargs*)

Channel post handler decorator.

Parameters

- **commands** –
- **regexp** –

- **func** –
- **content_types** –
- **kwargs** –

Returns

chat_join_request_handler(*func=None*, ***kwargs*)

chat_join_request handler.

Parameters

- **func** –
- **kwargs** –

Returns

chat_member_handler(*func=None*, ***kwargs*)

chat_member handler.

Parameters

- **func** –
- **kwargs** –

Returns

chosen_inline_handler(*func*, ***kwargs*)

Description: TBD

Parameters

- **func** –
- **kwargs** –

Returns

async close() → bool

Use this method to close the bot instance before moving it from one local server to another. You need to delete the webhook before calling this method to ensure that the bot isn't launched again after server restart. The method will return error 429 in the first 10 minutes after the bot is launched. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#close

async close_session()

Closes existing session of aiohttp. Use this function if you stop polling.

async copy_message(*chat_id: Union[int, str]*, *from_chat_id: Union[int, str]*, *message_id: int*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*, *ReplyKeyboardMarkup*, *ReplyKeyboardRemove*, *ForceReply*]] = None*, *timeout: Optional[int] = None*) → *MessageID*

Use this method to copy messages of any kind.

Telegram documentation: https://core.telegram.org/bots/api#copymessage

Parameters

- **chat_id** – which chat to forward
- **from_chat_id** – which chat message from
- **message_id** – message id
- **caption** –
- **parse_mode** –
- **caption_entities** –
- **disable_notification** –
- **reply_to_message_id** –
- **allow_sending_without_reply** –
- **reply_markup** –
- **timeout** –
- **protect_content** –

Returns
API reply.

async create_chat_invite_link(*chat_id: Union[int, str]*, *name: Optional[str] = None*, *expire_date: Optional[Union[int, datetime]] = None*, *member_limit: Optional[int] = None*, *creates_join_request: Optional[bool] = None*) → *ChatInviteLink*

Use this method to create an additional invite link for a chat. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#createchatinvitelink

Parameters

- **chat_id** – Id: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **name** – Invite link name; 0-32 characters
- **expire_date** – Point in time (Unix timestamp) when the link will expire
- **member_limit** – Maximum number of users that can be members of the chat simultaneously
- **creates_join_request** – True, if users joining the chat via the link need to be approved by chat administrators. If True, member_limit can't be specified

Returns

async create_invoice_link(*title: str*, *description: str*, *payload: str*, *provider_token: str*, *currency: str*, *prices: List[LabeledPrice]*, *max_tip_amount: Optional[int] = None*, *suggested_tip_amounts: Optional[List[int]] = None*, *provider_data: Optional[str] = None*, *photo_url: Optional[str] = None*, *photo_size: Optional[int] = None*, *photo_width: Optional[int] = None*, *photo_height: Optional[int] = None*, *need_name: Optional[bool] = None*, *need_phone_number: Optional[bool] = None*, *need_email: Optional[bool] = None*, *need_shipping_address: Optional[bool] = None*, *send_phone_number_to_provider: Optional[bool] = None*, *send_email_to_provider: Optional[bool] = None*, *is_flexible: Optional[bool] = None*) → str

Use this method to create a link for an invoice. Returns the created invoice link as String on success.

Telegram documentation: https://core.telegram.org/bots/api#createinvoicelink

Parameters

- **title** – Product name, 1-32 characters
- **description** – Product description, 1-255 characters
- **payload** – Bot-defined invoice payload, 1-128 bytes. This will not be displayed to the user, use for your internal processes.
- **provider_token** – Payments provider token, obtained via @Botfather
- **currency** – Three-letter ISO 4217 currency code, see https://core.telegram.org/bots/payments#supported-currencies
- **prices** – Price breakdown, a list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.)
- **max_tip_amount** – The maximum accepted amount for tips in the smallest units of the currency
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the smallest
- **provider_data** – A JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods
- **photo_size** – Photo size in bytes
- **photo_width** – Photo width
- **photo_height** – Photo height
- **need_name** – Pass True, if you require the user's full name to complete the order
- **need_phone_number** – Pass True, if you require the user's phone number to complete the order
- **need_email** – Pass True, if you require the user's email to complete the order
- **need_shipping_address** – Pass True, if you require the user's shipping address to complete the order
- **send_phone_number_to_provider** – Pass True, if user's phone number should be sent to provider
- **send_email_to_provider** – Pass True, if user's email address should be sent to provider
- **is_flexible** – Pass True, if the final price depends on the shipping method

Returns

Created invoice link as String on success.

async create_new_sticker_set(*user_id: int*, *name: str*, *title: str*, *emojis: str*, *png_sticker: Optional[Union[Any, str]] = None*, *tgs_sticker: Optional[Union[Any, str]] = None*, *webm_sticker: Optional[Union[Any, str]] = None*, *contains_masks: Optional[bool] = None*, *mask_position: Optional[MaskPosition] = None*) → bool

Use this method to create new sticker set owned by a user. The bot will be able to edit the created sticker set. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#createnewstickerset

Parameters

- **user_id** –
- **name** –
- **title** –
- **emojis** –
- **png_sticker** –
- **tgs_sticker** –
- **contains_masks** –
- **mask_position** –

Webm_sticker
Returns

async decline_chat_join_request(*chat_id: Union[str, int]*, *user_id: Union[int, str]*) → bool

Use this method to decline a chat join request. The bot must be an administrator in the chat for this to work and must have the can_invite_users administrator right. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#declinechatjoinrequest

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **user_id** – Unique identifier of the target user

Returns
True on success.

async delete_chat_photo(*chat_id: Union[int, str]*) → bool

Use this method to delete a chat photo. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success. Note: In regular groups (non-supergroups), this method will only work if the 'All Members Are Admins' setting is off in the target group.

Telegram documentation: https://core.telegram.org/bots/api#deletechatphoto

Parameters
chat_id – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)

async delete_chat_sticker_set(*chat_id: Union[int, str]*) → bool

Use this method to delete a group sticker set from a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Use the field can_set_sticker_set optionally returned in getChat requests to check if the bot can use this method. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletechatstickerset

Parameters
chat_id – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)

Returns
API reply.

async delete_message(*chat_id: Union[int, str]*, *message_id: int*, *timeout: Optional[int] = None*) → bool

Use this method to delete message. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletemessage

Parameters

- **chat_id** – in which chat to delete
- **message_id** – which message to delete
- **timeout** –

Returns
API reply.

async delete_my_commands(*scope: Optional[BotCommandScope] = None*, *language_code: Optional[int] = None*) → bool

Use this method to delete the list of the bot's commands for the given scope and user language. After deletion, higher level commands will be shown to affected users. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletemycommands

Parameters

- **scope** – The scope of users for which the commands are relevant. async defaults to BotCommandScopeasync default.
- **language_code** – A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

async delete_state(*user_id: int*, *chat_id: Optional[int] = None*)

Delete the current state of a user.

Parameters

- **user_id** –
- **chat_id** –

Returns

async delete_sticker_from_set(*sticker: str*) → bool

Use this method to delete a sticker from a set created by the bot. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#deletestickerfromset

Parameters
sticker –

Returns

async delete_webhook(*drop_pending_updates=None*, *timeout=None*)

Use this method to remove webhook integration if you decide to switch back to getUpdates.

Telegram documentation: https://core.telegram.org/bots/api#deletewebhook

Parameters

- **drop_pending_updates** – Pass True to drop all pending updates
- **timeout** – Integer. Request connection timeout

Returns
bool

async **download_file**(*file_path: str*) → bytes

async **edit_chat_invite_link**(*chat_id: Union[int, str]*, *invite_link: Optional[str] = None*, *name: Optional[str] = None*, *expire_date: Optional[Union[int, datetime]] = None*, *member_limit: Optional[int] = None*, *creates_join_request: Optional[bool] = None*) → *ChatInviteLink*

Use this method to edit a non-primary invite link created by the bot. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#editchatinvitelink

Parameters

- **chat_id** – Id: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **name** – Invite link name; 0-32 characters
- **invite_link** – The invite link to edit
- **expire_date** – Point in time (Unix timestamp) when the link will expire
- **member_limit** – Maximum number of users that can be members of the chat simultaneously
- **creates_join_request** – True, if users joining the chat via the link need to be approved by chat administrators. If True, member_limit can't be specified

Returns

async **edit_message_caption**(*caption: str*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*, *ReplyKeyboardMarkup*, *ReplyKeyboardRemove*, *ForceReply*]] = None*) → Union[*Message*, bool]

Use this method to edit captions of messages.

Telegram documentation: https://core.telegram.org/bots/api#editmessagecaption

Parameters

- **caption** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **parse_mode** –
- **caption_entities** –
- **reply_markup** –

Returns

async edit_message_live_location(*latitude: float*, *longitude: float*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*) → *Message*

Use this method to edit live location.

Telegram documentation: https://core.telegram.org/bots/api#editmessagelivelocation

Parameters

- **latitude** –
- **longitude** –
- **chat_id** –
- **message_id** –
- **reply_markup** –
- **timeout** –
- **inline_message_id** –
- **horizontal_accuracy** –
- **heading** –
- **proximity_alert_radius** –

Returns

async edit_message_media(*media: Any*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*) → Union[*Message*, bool]

Use this method to edit animation, audio, document, photo, or video messages. If a message is a part of a message album, then it can be edited only to a photo or a video. Otherwise, message type can be changed arbitrarily. When inline message is edited, new file can't be uploaded. Use previously uploaded file via its file_id or specify a URL.

Telegram documentation: https://core.telegram.org/bots/api#editmessagemedia

Parameters

- **media** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **reply_markup** –

Returns

async edit_message_reply_markup(*chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*) → Union[*Message*, bool]

Use this method to edit only the reply markup of messages.

Telegram documentation: https://core.telegram.org/bots/api#editmessagereplymarkup

Parameters

- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **reply_markup** –

Returns

async edit_message_text(*text: str*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *parse_mode: Optional[str] = None*, *entities: Optional[List[*MessageEntity*]] = None*, *disable_web_page_preview: Optional[bool] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*) → Union[*Message*, bool]

Use this method to edit text and game messages.

Telegram documentation: https://core.telegram.org/bots/api#editmessagetext

Parameters

- **text** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **parse_mode** –
- **entities** –
- **disable_web_page_preview** –
- **reply_markup** –

Returns

edited_channel_post_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, ***kwargs*)

Edit channel post handler decorator.

Parameters

- **commands** –
- **regexp** –
- **func** –
- **content_types** –
- **kwargs** –

Returns

edited_message_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, *chat_types=None*, ***kwargs*)

Edit message handler decorator.

Parameters

- **commands** –
- **regexp** –
- **func** –
- **content_types** –
- **chat_types** – list of chat types
- **kwargs** –

Returns

enable_saving_states(*filename='./.state-save/states.pkl'*)

Enable saving states (by default saving disabled)

Parameters
filename – Filename of saving file

async export_chat_invite_link(*chat_id: Union[int, str]*) → str

Use this method to export an invite link to a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#exportchatinvitelink

Parameters
chat_id – Id: Unique identifier for the target chat or username of the target channel (in the format @channelusername)

Returns
exported invite link as String on success.

async forward_message(*chat_id: Union[int, str]*, *from_chat_id: Union[int, str]*, *message_id: int*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to forward messages of any kind.

Telegram documentation: https://core.telegram.org/bots/api#forwardmessage

Parameters

- **disable_notification** –
- **chat_id** – which chat to forward
- **from_chat_id** – which chat message from
- **message_id** – message id
- **protect_content** –
- **timeout** –

Returns
API reply.

async get_chat(*chat_id: Union[int, str]*) → *Chat*

Use this method to get up to date information about the chat (current name of the user for one-on-one conversations, current username of a user, group or channel, etc.). Returns a Chat object on success.

Telegram documentation: https://core.telegram.org/bots/api#getchat

Parameters
chat_id –

Returns

async get_chat_administrators(*chat_id: Union[int, str]*) → List[*ChatMember*]

Use this method to get a list of administrators in a chat. On success, returns an Array of ChatMember objects that contains information about all chat administrators except other bots.

Telegram documentation: https://core.telegram.org/bots/api#getchatadministrators

Parameters
chat_id – Unique identifier for the target chat or username of the target supergroup or channel (in the format @channelusername)

Returns
API reply.

async get_chat_member(*chat_id: Union[int, str]*, *user_id: int*) → *ChatMember*

Use this method to get information about a member of a chat. Returns a ChatMember object on success.

Telegram documentation: https://core.telegram.org/bots/api#getchatmember

Parameters

- **chat_id** –
- **user_id** –

Returns
API reply.

async get_chat_member_count(*chat_id: Union[int, str]*) → int

Use this method to get the number of members in a chat. Returns Int on success.

Telegram documentation: https://core.telegram.org/bots/api#getchatmembercount

Parameters
chat_id –

Returns

get_chat_members_count(***kwargs*)

async get_chat_menu_button(*chat_id: Optional[Union[int, str]] = None*) → *MenuButton*

Use this method to get the current value of the bot's menu button in a private chat, or the default menu button. Returns MenuButton on success.

Telegram Documentation: https://core.telegram.org/bots/api#getchatmenubutton

Parameters
chat_id – Unique identifier for the target private chat. If not specified, default bot's menu button will be returned.

Returns
types.MenuButton

async get_file(*file_id: str*) → *File*

Use this method to get basic info about a file and prepare it for downloading. For the moment, bots can download files of up to 20MB in size. On success, a File object is returned. It is guaranteed that the link will be valid for at least 1 hour. When the link expires, a new one can be requested by calling get_file again.

Telegram documentation: https://core.telegram.org/bots/api#getfile

Parameters
file_id –

async get_file_url(*file_id: str*) → str

async get_game_high_scores(*user_id: int*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*) → List[*GameHighScore*]

Gets top points and game play.

Telegram documentation: https://core.telegram.org/bots/api#getgamehighscores

Parameters

- **user_id** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –

Returns

async get_me() → *User*

Returns basic information about the bot in form of a User object.

Telegram documentation: https://core.telegram.org/bots/api#getme

async get_my_commands(*scope: Optional[*BotCommandScope*]*, *language_code: Optional[str]*) → List[*BotCommand*]

Use this method to get the current list of the bot's commands. Returns List of BotCommand on success.

Telegram documentation: https://core.telegram.org/bots/api#getmycommands

Parameters

- **scope** – The scope of users for which the commands are relevant. async defaults to BotCommandScopeasync default.
- **language_code** – A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

async get_my_default_administrator_rights(*for_channels: Optional[bool] = None*) → *ChatAdministratorRights*

Use this method to get the current default administrator rights of the bot. Returns ChatAdministratorRights on success.

Telegram documentation: https://core.telegram.org/bots/api#getmydefaultadministratorrights

Parameters
for_channels – Pass True to get the default administrator rights of the bot in channels. Otherwise, the default administrator rights of the bot for groups and supergroups will be returned.

Returns
types.ChatAdministratorRights

async get_state(*user_id*, *chat_id: Optional[int] = None*)

Get current state of a user.

Parameters

- **user_id** –
- **chat_id** –

Returns
state of a user

async get_sticker_set(*name: str*) → *StickerSet*

Use this method to get a sticker set. On success, a StickerSet object is returned.

Telegram documentation: https://core.telegram.org/bots/api#getstickerset

Parameters
name –

Returns

async get_updates(*offset: Optional[int] = None*, *limit: Optional[int] = None*, *timeout: Optional[int] = 20*, *allowed_updates: Optional[List] = None*, *request_timeout: Optional[int] = None*) → List[*Update*]

Use this method to receive incoming updates using long polling (wiki). An Array of Update objects is returned.

Telegram documentation: https://core.telegram.org/bots/api#making-requests

Parameters

- **allowed_updates** – Array of string. List the types of updates you want your bot to receive.
- **offset** – Integer. Identifier of the first update to be returned.
- **limit** – Integer. Limits the number of updates to be retrieved.
- **timeout** – Integer. Request connection timeout
- **request_timeout** – Timeout in seconds for a request.

Returns
array of Updates

async get_user_profile_photos(*user_id: int*, *offset: Optional[int] = None*, *limit: Optional[int] = None*) → *UserProfilePhotos*

Retrieves the user profile photos of the person with 'user_id'

Telegram documentation: https://core.telegram.org/bots/api#getuserprofilephotos

Parameters

- **user_id** –
- **offset** –
- **limit** –

Returns
API reply.

async get_webhook_info(*timeout=None*)

Use this method to get current webhook status. Requires no parameters. If the bot is using getUpdates, will return an object with the url field empty.

Telegram documentation: https://core.telegram.org/bots/api#getwebhookinfo

Parameters
timeout – Integer. Request connection timeout

Returns
On success, returns a WebhookInfo object.

async infinity_polling(*timeout: int = 20*, *skip_pending: bool = False*, *request_timeout: int = 20*, *logger_level=40*, *allowed_updates: Optional[List[str]] = None*, **args*, ***kwargs*)

Wrap polling with infinite loop and exception handling to avoid bot stops polling.

Parameters

- **timeout** – Request connection timeout
- **request_timeout** – Timeout in seconds for long polling (see API docs)
- **skip_pending** – skip old updates
- **logger_level** – Custom logging level for infinity_polling logging. Use logger levels from logging as a value. None/NOTSET = no error logging
- **allowed_updates** – A list of the update types you want your bot to receive. For example, specify [“message”, “edited_channel_post”, “callback_query”] to only receive updates of these types. See util.update_types for a complete list of available update types. Specify an empty list to receive all update types except chat_member (default). If not specified, the previous setting will be used.

  Please note that this parameter doesn't affect updates created before the call to the get_updates, so unwanted updates may be received for a short period of time.

inline_handler(*func*, ***kwargs*)

Inline call handler decorator.

Parameters

- **func** –
- **kwargs** –

Returns

async kick_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *until_date: Optional[Union[int, datetime]] = None*, *revoke_messages: Optional[bool] = None*) → bool

This function is deprecated. Use *ban_chat_member* instead

async leave_chat(*chat_id: Union[int, str]*) → bool

Use this method for your bot to leave a group, supergroup or channel. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#leavechat

Parameters
chat_id –

Returns

async log_out() → bool

Use this method to log out from the cloud Bot API server before launching the bot locally. You MUST log out the bot before running it locally, otherwise there is no guarantee that the bot will receive updates. After a successful call, you can immediately log in on a local server, but will not be able to log in back to the cloud Bot API server for 10 minutes. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#logout

message_handler(*commands=None*, *regexp=None*, *func=None*, *content_types=None*, *chat_types=None*, ***kwargs*)

Message handler decorator. This decorator can be used to decorate functions that must handle certain types of messages. All message handlers are tested in the order they were added.

Example:

```
bot = TeleBot('TOKEN')

# Handles all messages which text matches regexp.
@bot.message_handler(regexp='someregexp')
async def command_help(message):
    bot.send_message(message.chat.id, 'Did someone call for help?')

# Handles messages in private chat
@bot.message_handler(chat_types=['private']) # You can add more chat types
async def command_help(message):
    bot.send_message(message.chat.id, 'Private chat detected, sir!')

# Handle all sent documents of type 'text/plain'.
@bot.message_handler(func=lambda message: message.document.mime_type == 'text/
↪plain',
    content_types=['document'])
async def command_handle_document(message):
    bot.send_message(message.chat.id, 'Document received, sir!')

# Handle all other messages.
@bot.message_handler(func=lambda message: True, content_types=['audio', 'photo',
↪ 'voice', 'video', 'document',
    'text', 'location', 'contact', 'sticker'])
async def async default_command(message):
    bot.send_message(message.chat.id, "This is the async default command␣
↪handler.")
```

Parameters

- **commands** – Optional list of strings (commands to handle).
- **regexp** – Optional regular expression.
- **func** – Optional lambda function. The lambda receives the message to test as the first parameter. It must return True if the command should handle the message.
- **content_types** – Supported message content types. Must be a list. async defaults to [‘text’].
- **chat_types** – list of chat types

my_chat_member_handler(*func=None*, ***kwargs*)

my_chat_member handler.

Parameters

- **func** –
- **kwargs** –

Returns

async pin_chat_message(*chat_id: Union[int, str]*, *message_id: int*, *disable_notification: Optional[bool] = False*) → bool

Use this method to pin a message in a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#pinchatmessage

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **message_id** – Int: Identifier of a message to pin
- **disable_notification** – Bool: Pass True, if it is not necessary to send a notification to all group members about the new pinned message

Returns

poll_answer_handler(*func=None*, ***kwargs*)

Poll_answer request handler.

Parameters

- **func** –
- **kwargs** –

Returns

poll_handler(*func*, ***kwargs*)

Poll request handler.

Parameters

- **func** –
- **kwargs** –

Returns

async polling(*non_stop: bool = False*, *skip_pending=False*, *interval: int = 0*, *timeout: int = 20*, *request_timeout: int = 20*, *allowed_updates: Optional[List[str]] = None*, *none_stop: Optional[bool] = None*)

This allows the bot to retrieve Updates automatically and notify listeners and message handlers accordingly.

Warning: Do not call this function more than once!

Always get updates.

Parameters

- **interval** – Delay between two update retrivals
- **non_stop** – Do not stop polling when an ApiException occurs.

- **timeout** – Request connection timeout
- **skip_pending** – skip old updates
- **request_timeout** – Timeout in seconds for a request.
- **allowed_updates** – A list of the update types you want your bot to receive. For example, specify [“message”, “edited_channel_post”, “callback_query”] to only receive updates of these types. See util.update_types for a complete list of available update types. Specify an empty list to receive all update types except chat_member (default). If not specified, the previous setting will be used.

  Please note that this parameter doesn't affect updates created before the call to the get_updates, so unwanted updates may be received for a short period of time.
- **none_stop** – Deprecated, use non_stop. Old typo f***up compatibility

Returns

pre_checkout_query_handler(*func*, ***kwargs*)

Pre-checkout request handler.

Parameters

- **func** –
- **kwargs** –

Returns

async process_chat_join_request(*chat_join_request*)

async process_middlewares(*update_type*)

async process_new_callback_query(*new_callback_querys*)

async process_new_channel_posts(*channel_post*)

async process_new_chat_member(*chat_members*)

async process_new_chosen_inline_query(*new_chosen_inline_querys*)

async process_new_edited_channel_posts(*edited_channel_post*)

async process_new_edited_messages(*edited_message*)

async process_new_inline_query(*new_inline_querys*)

async process_new_messages(*new_messages*)

async process_new_my_chat_member(*my_chat_members*)

async process_new_poll(*polls*)

async process_new_poll_answer(*poll_answers*)

async process_new_pre_checkout_query(*pre_checkout_querys*)

async process_new_shipping_query(*new_shipping_querys*)

async process_new_updates(*updates*)

Process new updates. Just pass list of updates - each update should be instance of Update object.

Parameters
updates – list of updates

async promote_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *can_change_info: Optional[bool] = None*, *can_post_messages: Optional[bool] = None*, *can_edit_messages: Optional[bool] = None*, *can_delete_messages: Optional[bool] = None*, *can_invite_users: Optional[bool] = None*, *can_restrict_members: Optional[bool] = None*, *can_pin_messages: Optional[bool] = None*, *can_promote_members: Optional[bool] = None*, *is_anonymous: Optional[bool] = None*, *can_manage_chat: Optional[bool] = None*, *can_manage_video_chats: Optional[bool] = None*, *can_manage_voice_chats: Optional[bool] = None*) → bool

Use this method to promote or demote a user in a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Pass False for all boolean parameters to demote a user.

Telegram documentation: https://core.telegram.org/bots/api#promotechatmember

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **user_id** – Int : Unique identifier of the target user
- **can_change_info** – Bool: Pass True, if the administrator can change chat title, photo and other settings
- **can_post_messages** – Bool : Pass True, if the administrator can create channel posts, channels only
- **can_edit_messages** – Bool : Pass True, if the administrator can edit messages of other users, channels only
- **can_delete_messages** – Bool : Pass True, if the administrator can delete messages of other users
- **can_invite_users** – Bool : Pass True, if the administrator can invite new users to the chat
- **can_restrict_members** – Bool: Pass True, if the administrator can restrict, ban or unban chat members
- **can_pin_messages** – Bool: Pass True, if the administrator can pin messages, supergroups only
- **can_promote_members** – Bool: Pass True, if the administrator can add new administrators with a subset of his own privileges or demote administrators that he has promoted, directly or indirectly (promoted by administrators that were appointed by him)
- **is_anonymous** – Bool: Pass True, if the administrator's presence in the chat is hidden
- **can_manage_chat** – Bool: Pass True, if the administrator can access the chat event log, chat statistics, message statistics in channels, see channel members, see anonymous administrators in supergroups and ignore slow mode. Implied by any other administrator privilege
- **can_manage_video_chats** – Bool: Pass True, if the administrator can manage voice chats For now, bots can use this privilege only for passing to other administrators.

- **can_manage_voice_chats** – Deprecated, use can_manage_video_chats

Returns
True on success.

register_callback_query_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)
Registers callback query handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

register_channel_post_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *pass_bot=False*, ***kwargs*)
Registers channel post message handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands
- **regexp** –
- **func** –

Returns
decorated function

register_chat_join_request_handler(*callback*, *func=None*, *pass_bot=False*, ***kwargs*)
Registers chat join request handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

register_chat_member_handler(*callback*, *func=None*, *pass_bot=False*, ***kwargs*)
Registers chat member handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

register_chosen_inline_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers chosen inline handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

register_edited_channel_post_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *pass_bot=False*, ***kwargs*)

Registers edited channel post message handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands
- **regexp** –
- **func** –

Returns
decorated function

register_edited_message_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *chat_types=None*, *pass_bot=False*, ***kwargs*)

Registers edited message handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands
- **regexp** –
- **func** –
- **chat_types** – True for private chat

Returns
decorated function

register_inline_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers inline handler.

Parameters

- **pass_bot** –
- **callback** – function to be called

- **func** –

Returns
decorated function

register_message_handler(*callback*, *content_types=None*, *commands=None*, *regexp=None*, *func=None*, *chat_types=None*, *pass_bot=False*, ***kwargs*)

Registers message handler.

Parameters

- **callback** – function to be called
- **content_types** – list of content_types
- **commands** – list of commands
- **regexp** –
- **func** –
- **chat_types** – True for private chat
- **pass_bot** – True if you want to get TeleBot instance in your handler

Returns
decorated function

register_my_chat_member_handler(*callback*, *func=None*, *pass_bot=False*, ***kwargs*)

Registers my chat member handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

register_poll_answer_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers poll answer handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

register_poll_handler(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers poll handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

`register_pre_checkout_query_handler`(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers pre-checkout request handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

`register_shipping_query_handler`(*callback*, *func*, *pass_bot=False*, ***kwargs*)

Registers shipping query handler.

Parameters

- **pass_bot** –
- **callback** – function to be called
- **func** –

Returns
decorated function

`async remove_webhook`()

Alternative for delete_webhook but uses set_webhook

`async reply_to`(*message: Message*, *text: str*, ***kwargs*) → *Message*

Convenience function for *send_message(message.chat.id, text, reply_to_message_id=message.message_id, **kwargs)*

Parameters

- **message** –
- **text** –
- **kwargs** –

Returns

`async reset_data`(*user_id: int*, *chat_id: Optional[int] = None*)

Reset data for a user in chat.

Parameters

- **user_id** –
- **chat_id** –

`async restrict_chat_member`(*chat_id: Union[int, str]*, *user_id: int*, *until_date: Optional[Union[int, datetime]] = None*, *can_send_messages: Optional[bool] = None*, *can_send_media_messages: Optional[bool] = None*, *can_send_polls: Optional[bool] = None*, *can_send_other_messages: Optional[bool] = None*, *can_add_web_page_previews: Optional[bool] = None*, *can_change_info: Optional[bool] = None*, *can_invite_users: Optional[bool] = None*, *can_pin_messages: Optional[bool] = None*) → bool

Use this method to restrict a user in a supergroup. The bot must be an administrator in the supergroup for this to work and must have the appropriate admin rights. Pass True for all boolean parameters to lift restrictions from a user.

Telegram documentation: https://core.telegram.org/bots/api#restrictchatmember

Parameters

- **chat_id** – Int or String : Unique identifier for the target group or username of the target supergroup or channel (in the format @channelusername)
- **user_id** – Int : Unique identifier of the target user
- **until_date** – Date when restrictions will be lifted for the user, unix time. If user is restricted for more than 366 days or less than 30 seconds from the current time, they are considered to be restricted forever
- **can_send_messages** – Pass True, if the user can send text messages, contacts, locations and venues
- **can_send_media_messages** – Pass True, if the user can send audios, documents, photos, videos, video notes and voice notes, implies can_send_messages
- **can_send_polls** – Pass True, if the user is allowed to send polls, implies can_send_messages
- **can_send_other_messages** – Pass True, if the user can send animations, games, stickers and use inline bots, implies can_send_media_messages
- **can_add_web_page_previews** – Pass True, if the user may add web page previews to their messages, implies can_send_media_messages
- **can_change_info** – Pass True, if the user is allowed to change the chat title, photo and other settings. Ignored in public supergroups
- **can_invite_users** – Pass True, if the user is allowed to invite new users to the chat, implies can_invite_users
- **can_pin_messages** – Pass True, if the user is allowed to pin messages. Ignored in public supergroups

Returns

True on success

retrieve_data(*user_id: int*, *chat_id: Optional[int] = None*)

async revoke_chat_invite_link(*chat_id: Union[int, str]*, *invite_link: str*) → *ChatInviteLink*

Use this method to revoke an invite link created by the bot. Note: If the primary link is revoked, a new link is automatically generated The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#revokechatinvitelink

Parameters

- **chat_id** – Id: Unique identifier for the target chat or username of the target channel(in the format @channelusername)
- **invite_link** – The invite link to revoke

Returns

API reply.

async send_animation(*chat_id: Union[int, str]*, *animation: Union[Any, str]*, *duration: Optional[int] = None*, *width: Optional[int] = None*, *height: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[MessageEntity]] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to send animation files (GIF or H.264/MPEG-4 AVC video without sound).

Telegram documentation: https://core.telegram.org/bots/api#sendanimation

Parameters

- **chat_id** – Integer : Unique identifier for the message recipient — User or GroupChat id
- **animation** – InputFile or String : Animation to send. You can either pass a file_id as String to resend an animation that is already on the Telegram server
- **duration** – Integer : Duration of sent video in seconds
- **width** – Integer : Video width
- **height** – Integer : Video height
- **thumb** – InputFile or String : Thumbnail of the file sent
- **caption** – String : Animation caption (may also be used when resending animation by file_id).
- **parse_mode** –
- **protect_content** –
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** –
- **timeout** –
- **caption_entities** –
- **allow_sending_without_reply** –

Returns

async send_audio(*chat_id: Union[int, str]*, *audio: Union[Any, str]*, *caption: Optional[str] = None*, *duration: Optional[int] = None*, *performer: Optional[str] = None*, *title: Optional[str] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *parse_mode: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption_entities: Optional[List[MessageEntity]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send audio files, if you want Telegram clients to display them in the music player. Your audio must be in the .mp3 format.

Telegram documentation: https://core.telegram.org/bots/api#sendaudio

Parameters

- **chat_id** – Unique identifier for the message recipient
- **audio** – Audio file to send.
- **caption** –
- **duration** – Duration of the audio in seconds
- **performer** – Performer
- **title** – Track name
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **reply_markup** –
- **parse_mode** –
- **disable_notification** –
- **timeout** –
- **thumb** –
- **caption_entities** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns
Message

async send_chat_action(*chat_id: Union[int, str]*, *action: str*, *timeout: Optional[int] = None*) → bool

Use this method when you need to tell the user that something is happening on the bot's side. The status is set for 5 seconds or less (when a message arrives from your bot, Telegram clients clear its typing status).

Telegram documentation: https://core.telegram.org/bots/api#sendchataction

Parameters

- **chat_id** –
- **action** – One of the following strings: 'typing', 'upload_photo', 'record_video', 'upload_video', 'record_audio', 'upload_audio', 'upload_document', 'find_location', 'record_video_note', 'upload_video_note'.
- **timeout** –

Returns
API reply. :type: boolean

async send_contact(*chat_id: Union[int, str]*, *phone_number: str*, *first_name: str*, *last_name: Optional[str] = None*, *vcard: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send phone contacts.

Telegram documentation: https://core.telegram.org/bots/api#sendcontact

Parameters

- **chat_id** – Integer or String : Unique identifier for the target chat or username of the target channel
- **phone_number** – String : Contact's phone number
- **first_name** – String : Contact's first name
- **last_name** – String : Contact's last name
- **vcard** – String : Additional data about the contact in the form of a vCard, 0-2048 bytes
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –
- **protect_content** –

async **send_dice**(*chat_id: Union[int, str]*, *emoji: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send dices.

Telegram documentation: https://core.telegram.org/bots/api#senddice

Parameters

- **chat_id** –
- **emoji** –
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

Message

async **send_document**(*chat_id: Union[int, str]*, *document: Union[Any, str]*, *reply_to_message_id: Optional[int] = None*, *caption: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *parse_mode: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *visible_file_name: Optional[str] = None*, *disable_content_type_detection: Optional[bool] = None*, *data: Optional[Union[Any, str]] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send general files.

Telegram documentation: https://core.telegram.org/bots/api#senddocument

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **document** – (document) File to send. Pass a file_id as String to send a file that exists on the Telegram servers (recommended), pass an HTTP URL as a String for Telegram to get a file from the Internet, or upload a new one using multipart/form-data
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **caption** – Document caption (may also be used when resending documents by file_id), 0-1024 characters after entities parsing
- **reply_markup** –
- **parse_mode** – Mode for parsing entities in the document caption
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **timeout** –
- **thumb** – InputFile or String : Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass "attach://<file_attach_name>" if the thumbnail was uploaded using multipart/form-data under <file_attach_name>
- **caption_entities** –
- **allow_sending_without_reply** –
- **visible_file_name** – allows to async define file name that will be visible in the Telegram instead of original file name
- **disable_content_type_detection** – Disables automatic server-side content type detection for files uploaded using multipart/form-data
- **data** – function typo compatibility: do not use it
- **protect_content** –

Returns

API reply.

async send_game(*chat_id: Union[int, str]*, *game_short_name: str*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Used to send the game.

Telegram documentation: https://core.telegram.org/bots/api#sendgame

Parameters

- **chat_id** –

- **game_short_name** –
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

async **send_invoice**(*chat_id: Union[int, str]*, *title: str*, *description: str*, *invoice_payload: str*, *provider_token: str*, *currency: str*, *prices: List[*LabeledPrice*]*, *start_parameter: Optional[str] = None*, *photo_url: Optional[str] = None*, *photo_size: Optional[int] = None*, *photo_width: Optional[int] = None*, *photo_height: Optional[int] = None*, *need_name: Optional[bool] = None*, *need_phone_number: Optional[bool] = None*, *need_email: Optional[bool] = None*, *need_shipping_address: Optional[bool] = None*, *send_phone_number_to_provider: Optional[bool] = None*, *send_email_to_provider: Optional[bool] = None*, *is_flexible: Optional[bool] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *provider_data: Optional[str] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *max_tip_amount: Optional[int] = None*, *suggested_tip_amounts: Optional[List[int]] = None*, *protect_content: Optional[bool] = None*) → *Message*

Sends invoice.

Telegram documentation: https://core.telegram.org/bots/api#sendinvoice

Parameters

- **chat_id** – Unique identifier for the target private chat
- **title** – Product name
- **description** – Product description
- **invoice_payload** – Bot-async defined invoice payload, 1-128 bytes. This will not be displayed to the user, use for your internal processes.
- **provider_token** – Payments provider token, obtained via @Botfather
- **currency** – Three-letter ISO 4217 currency code, see https://core.telegram.org/bots/payments#supported-currencies
- **prices** – Price breakdown, a list of components (e.g. product price, tax, discount, delivery cost, delivery tax, bonus, etc.)
- **start_parameter** – Unique deep-linking parameter that can be used to generate this invoice when used as a start parameter
- **photo_url** – URL of the product photo for the invoice. Can be a photo of the goods or a marketing image for a service. People like it better when they see what they are paying for.
- **photo_size** – Photo size
- **photo_width** – Photo width
- **photo_height** – Photo height

- **need_name** – Pass True, if you require the user's full name to complete the order
- **need_phone_number** – Pass True, if you require the user's phone number to complete the order
- **need_email** – Pass True, if you require the user's email to complete the order
- **need_shipping_address** – Pass True, if you require the user's shipping address to complete the order
- **is_flexible** – Pass True, if the final price depends on the shipping method
- **send_phone_number_to_provider** – Pass True, if user's phone number should be sent to provider
- **send_email_to_provider** – Pass True, if user's email address should be sent to provider
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **reply_markup** – A JSON-serialized object for an inline keyboard. If empty, one 'Pay total price' button will be shown. If not empty, the first button must be a Pay button
- **provider_data** – A JSON-serialized data about the invoice, which will be shared with the payment provider. A detailed description of required fields should be provided by the payment provider.
- **timeout** –
- **allow_sending_without_reply** –
- **max_tip_amount** – The maximum accepted amount for tips in the smallest units of the currency
- **suggested_tip_amounts** – A JSON-serialized array of suggested amounts of tips in the smallest units of the currency. At most 4 suggested tip amounts can be specified. The suggested tip amounts must be positive, passed in a strictly increased order and must not exceed max_tip_amount.
- **protect_content** –

Returns

async send_location(*chat_id: Union[int, str]*, *latitude: float*, *longitude: float*, *live_period: Optional[int] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *horizontal_accuracy: Optional[float] = None*, *heading: Optional[int] = None*, *proximity_alert_radius: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send point on the map.

Telegram documentation: https://core.telegram.org/bots/api#sendlocation

Parameters

- **chat_id** –
- **latitude** –
- **longitude** –

- **live_period** –
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** –
- **timeout** –
- **horizontal_accuracy** –
- **heading** –
- **proximity_alert_radius** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns
API reply.

async **send_media_group**(*chat_id: Union[int, str]*, *media: List[Union[*InputMediaAudio, InputMediaDocument, InputMediaPhoto, InputMediaVideo*]]*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*) → List[*Message*]

send a group of photos or videos as an album. On success, an array of the sent Messages is returned.

Telegram documentation: https://core.telegram.org/bots/api#sendmediagroup

Parameters

- **chat_id** –
- **media** –
- **disable_notification** –
- **reply_to_message_id** –
- **timeout** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns

async **send_message**(*chat_id: Union[int, str]*, *text: str*, *parse_mode: Optional[str] = None*, *entities: Optional[List[*MessageEntity*]] = None*, *disable_web_page_preview: Optional[bool] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to send text messages.

Warning: Do not send more than about 4000 characters each message, otherwise you'll risk an HTTP 414 error. If you must send more than 4000 characters, use the *split_string* or *smart_split* function in util.py.

Telegram documentation: https://core.telegram.org/bots/api#sendmessage

Parameters

- **chat_id** –
- **text** –
- **disable_web_page_preview** –
- **reply_to_message_id** –
- **reply_markup** –
- **parse_mode** –
- **disable_notification** – Boolean, Optional. Sends the message silently.
- **timeout** –
- **entities** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns
API reply.

async **send_photo**(*chat_id: Union[int, str]*, *photo: Union[Any, str]*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to send photos.

Telegram documentation: https://core.telegram.org/bots/api#sendphoto

Parameters

- **chat_id** –
- **photo** –
- **caption** –
- **parse_mode** –
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **caption_entities** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns
API reply.

async send_poll(*chat_id: Union[int, str]*, *question: str*, *options: List[str]*, *is_anonymous: Optional[bool] = None*, *type: Optional[str] = None*, *allows_multiple_answers: Optional[bool] = None*, *correct_option_id: Optional[int] = None*, *explanation: Optional[str] = None*, *explanation_parse_mode: Optional[str] = None*, *open_period: Optional[int] = None*, *close_date: Optional[Union[int, datetime]] = None*, *is_closed: Optional[bool] = None*, *disable_notification: Optional[bool] = False*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *timeout: Optional[int] = None*, *explanation_entities: Optional[List[*MessageEntity*]] = None*, *protect_content: Optional[bool] = None*) → *Message*

Send polls.

Telegram documentation: https://core.telegram.org/bots/api#sendpoll

Parameters

- **chat_id** –
- **question** –
- **options** – array of str with answers
- **is_anonymous** –
- **type** –
- **allows_multiple_answers** –
- **correct_option_id** –
- **explanation** –
- **explanation_parse_mode** –
- **open_period** –
- **close_date** –
- **is_closed** –
- **disable_notification** –
- **reply_to_message_id** –
- **allow_sending_without_reply** –
- **reply_markup** –
- **timeout** –
- **explanation_entities** –
- **protect_content** –

Returns

async send_sticker(*chat_id: Union[int, str]*, *sticker: Union[Any, str]*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *data: Optional[Union[Any, str]] = None*) → *Message*

Use this method to send .webp stickers.

Telegram documentation: https://core.telegram.org/bots/api#sendsticker

Parameters

- **chat_id** –
- **sticker** –
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** – to disable the notification
- **timeout** – timeout
- **allow_sending_without_reply** –
- **protect_content** –
- **data** – deprecated, for backward compatibility

Returns
API reply.

async **send_venue**(*chat_id: Union[int, str]*, *latitude: float*, *longitude: float*, *title: str*, *address: str*, *foursquare_id: Optional[str] = None*, *foursquare_type: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*, *timeout: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *google_place_id: Optional[str] = None*, *google_place_type: Optional[str] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send information about a venue.

Telegram documentation: https://core.telegram.org/bots/api#sendvenue

Parameters

- **chat_id** – Integer or String : Unique identifier for the target chat or username of the target channel
- **latitude** – Float : Latitude of the venue
- **longitude** – Float : Longitude of the venue
- **title** – String : Name of the venue
- **address** – String : Address of the venue
- **foursquare_id** – String : Foursquare identifier of the venue
- **foursquare_type** – Foursquare type of the venue, if known. (For example, “arts_entertainment/async default”, “arts_entertainment/aquarium” or “food/icecream”.)
- **disable_notification** –
- **reply_to_message_id** –
- **reply_markup** –
- **timeout** –
- **allow_sending_without_reply** –
- **google_place_id** –
- **google_place_type** –

- **protect_content** –

Returns

async **send_video**(*chat_id: Union[int, str]*, *video: Union[Any, str]*, *duration: Optional[int] = None*, *width: Optional[int] = None*, *height: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *caption: Optional[str] = None*, *parse_mode: Optional[str] = None*, *caption_entities: Optional[List[MessageEntity]] = None*, *supports_streaming: Optional[bool] = None*, *disable_notification: Optional[bool] = None*, *protect_content: Optional[bool] = None*, *reply_to_message_id: Optional[int] = None*, *allow_sending_without_reply: Optional[bool] = None*, *reply_markup: Optional[Union[InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply]] = None*, *timeout: Optional[int] = None*, *data: Optional[Union[Any, str]] = None*) → *Message*

Use this method to send video files, Telegram clients support mp4 videos (other formats may be sent as Document).

Telegram documentation: https://core.telegram.org/bots/api#sendvideo

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **video** – Video to send. You can either pass a file_id as String to resend a video that is already on the Telegram servers, or upload a new video file using multipart/form-data.
- **duration** – Duration of sent video in seconds
- **width** – Video width
- **height** – Video height
- **thumb** – Thumbnail of the file sent; can be ignored if thumbnail generation for the file is supported server-side. The thumbnail should be in JPEG format and less than 200 kB in size. A thumbnail's width and height should not exceed 320. Ignored if the file is not uploaded using multipart/form-data. Thumbnails can't be reused and can be only uploaded as a new file, so you can pass "attach://<file_attach_name>" if the thumbnail was uploaded using multipart/form-data under <file_attach_name>.
- **caption** – Video caption (may also be used when resending videos by file_id), 0-1024 characters after entities parsing
- **parse_mode** – Mode for parsing entities in the video caption
- **caption_entities** –
- **supports_streaming** – Pass True, if the uploaded video is suitable for streaming
- **disable_notification** – Sends the message silently. Users will receive a notification with no sound.
- **protect_content** –
- **reply_to_message_id** – If the message is a reply, ID of the original message
- **allow_sending_without_reply** –
- **reply_markup** –
- **timeout** –
- **data** – deprecated, for backward compatibility

async send_video_note(*chat_id: Union[int, str]*, *data: Union[Any, str]*, *duration: Optional[int] = None*, *length: Optional[int] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *thumb: Optional[Union[Any, str]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

As of v.4.0, Telegram clients support rounded square mp4 videos of up to 1 minute long. Use this method to send
video messages.

Telegram documentation: https://core.telegram.org/bots/api#sendvideonote

Parameters

- **chat_id** – Integer : Unique identifier for the message recipient — User or GroupChat id
- **data** – InputFile or String : Video note to send. You can either pass a file_id as String to resend a video that is already on the Telegram server
- **duration** – Integer : Duration of sent video in seconds
- **length** – Integer : Video width and height, Can't be None and should be in range of (0, 640)
- **reply_to_message_id** –
- **reply_markup** –
- **disable_notification** –
- **timeout** –
- **thumb** – InputFile or String : Thumbnail of the file sent
- **allow_sending_without_reply** –
- **protect_content** –

Returns

async send_voice(*chat_id: Union[int, str]*, *voice: Union[Any, str]*, *caption: Optional[str] = None*, *duration: Optional[int] = None*, *reply_to_message_id: Optional[int] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *parse_mode: Optional[str] = None*, *disable_notification: Optional[bool] = None*, *timeout: Optional[int] = None*, *caption_entities: Optional[List[*MessageEntity*]] = None*, *allow_sending_without_reply: Optional[bool] = None*, *protect_content: Optional[bool] = None*) → *Message*

Use this method to send audio files, if you want Telegram clients to display the file as a playable voice message.

Telegram documentation: https://core.telegram.org/bots/api#sendvoice

Parameters

- **chat_id** – Unique identifier for the message recipient.
- **voice** –
- **caption** –
- **duration** – Duration of sent audio in seconds
- **reply_to_message_id** –

- **reply_markup** –
- **parse_mode** –
- **disable_notification** –
- **timeout** –
- **caption_entities** –
- **allow_sending_without_reply** –
- **protect_content** –

Returns
Message

async **set_chat_administrator_custom_title**(*chat_id: Union[int, str]*, *user_id: int*, *custom_title: str*) → bool

Use this method to set a custom title for an administrator in a supergroup promoted by the bot.

Telegram documentation: https://core.telegram.org/bots/api#setchatadministratorcustomtitle

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **user_id** – Unique identifier of the target user
- **custom_title** – New custom title for the administrator; 0-16 characters, emoji are not allowed

Returns
True on success.

async **set_chat_description**(*chat_id: Union[int, str]*, *description: Optional[str] = None*) → bool

Use this method to change the description of a supergroup or a channel. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights.

Telegram documentation: https://core.telegram.org/bots/api#setchatdescription

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **description** – Str: New chat description, 0-255 characters

Returns
True on success.

async **set_chat_menu_button**(*chat_id: Optional[Union[int, str]] = None*, *menu_button: Optional[MenuButton] = None*) → bool

Use this method to change the bot's menu button in a private chat, or the default menu button. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setchatmenubutton

Parameters

- **chat_id** – Unique identifier for the target private chat. If not specified, default bot's menu button will be changed.
- **menu_button** – A JSON-serialized object for the new bot's menu button. Defaults to MenuButtonDefault

async set_chat_permissions(*chat_id: Union[int, str]*, *permissions:* ChatPermissions) → bool

Use this method to set async default chat permissions for all members. The bot must be an administrator in the group or a supergroup for this to work and must have the can_restrict_members admin rights.

Telegram documentation: https://core.telegram.org/bots/api#setchatpermissions

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **permissions** – New async default chat permissions

Returns
True on success

async set_chat_photo(*chat_id: Union[int, str]*, *photo: Any*) → bool

Use this method to set a new profile photo for the chat. Photos can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success. Note: In regular groups (non-supergroups), this method will only work if the 'All Members Are Admins' setting is off in the target group.

Telegram documentation: https://core.telegram.org/bots/api#setchatphoto

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **photo** – InputFile: New chat photo, uploaded using multipart/form-data

Returns

async set_chat_sticker_set(*chat_id: Union[int, str]*, *sticker_set_name: str*) → *StickerSet*

Use this method to set a new group sticker set for a supergroup. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Use the field can_set_sticker_set optionally returned in getChat requests to check if the bot can use this method. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setchatstickerset

Parameters

- **chat_id** – Unique identifier for the target chat or username of the target supergroup (in the format @supergroupusername)
- **sticker_set_name** – Name of the sticker set to be set as the group sticker set

Returns
API reply.

async set_chat_title(*chat_id: Union[int, str]*, *title: str*) → bool

Use this method to change the title of a chat. Titles can't be changed for private chats. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success. Note: In regular groups (non-supergroups), this method will only work if the 'All Members Are Admins' setting is off in the target group.

Telegram documentation: https://core.telegram.org/bots/api#setchattitle

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **title** – New chat title, 1-255 characters

Returns

async set_game_score(*user_id: Union[int, str]*, *score: int*, *force: Optional[bool] = None*, *chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *disable_edit_message: Optional[bool] = None*) → Union[*Message*, bool]

Sets the value of points in the game to a specific user.

Telegram documentation: https://core.telegram.org/bots/api#setgamescore

Parameters

- **user_id** –
- **score** –
- **force** –
- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **disable_edit_message** –

Returns

async set_my_commands(*commands: List[BotCommand]*, *scope: Optional[BotCommandScope] = None*, *language_code: Optional[str] = None*) → bool

Use this method to change the list of the bot's commands.

Telegram documentation: https://core.telegram.org/bots/api#setmycommands

Parameters

- **commands** – List of BotCommand. At most 100 commands can be specified.
- **scope** – The scope of users for which the commands are relevant. async defaults to BotCommandScopeasync default.
- **language_code** – A two-letter ISO 639-1 language code. If empty, commands will be applied to all users from the given scope, for whose language there are no dedicated commands

Returns

async set_my_default_administrator_rights(*rights: Optional[ChatAdministratorRights] = None*, *for_channels: Optional[bool] = None*) → bool

Use this method to change the default administrator rights requested by the bot when it's added as an administrator to groups or channels. These rights will be suggested to users, but they are are free to modify the list before adding the bot. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setmydefaultadministratorrights

Parameters

- **rights** – A JSON-serialized object describing new default administrator rights. If not specified, the default administrator rights will be cleared.
- **for_channels** – Pass True to change the default administrator rights of the bot in channels. Otherwise, the default administrator rights of the bot for groups and supergroups will be changed.

async set_state(*user_id: int*, *state: Union[State, int, str]*, *chat_id: Optional[int] = None*)

Sets a new state of a user.

Parameters

- **user_id** –
- **chat_id** –
- **state** – new state. can be string or integer.

async set_sticker_position_in_set(*sticker: str*, *position: int*) → bool

Use this method to move a sticker in a set created by the bot to a specific position . Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setstickerpositioninset

Parameters

- **sticker** –
- **position** –

Returns

async set_sticker_set_thumb(*name: str*, *user_id: int*, *thumb: Optional[Union[Any, str]] = None*)

Use this method to set the thumbnail of a sticker set. Animated thumbnails can be set for animated sticker sets only. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setstickersetthumb

Parameters

- **name** – Sticker set name
- **user_id** – User identifier
- **thumb** – A PNG image with the thumbnail, must be up to 128 kilobytes in size and have width and height exactly 100px, or a TGS animation with the thumbnail up to 32 kilobytes in size; see https://core.telegram.org/animated_stickers#technical-requirements

set_update_listener(*func*)

Update listener is a function that gets any update.

Parameters
func – function that should get update.

async set_webhook(*url=None*, *certificate=None*, *max_connections=None*, *allowed_updates=None*, *ip_address=None*, *drop_pending_updates=None*, *timeout=None*, *secret_token=None*)

Use this method to specify a url and receive incoming updates via an outgoing webhook. Whenever there is an update for the bot, we will send an HTTPS POST request to the specified url, containing a JSON-serialized Update. In case of an unsuccessful request, we will give up after a reasonable amount of attempts. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#setwebhook

Parameters

- **url** – HTTPS url to send updates to. Use an empty string to remove webhook integration
- **certificate** – Upload your public key certificate so that the root certificate in use can be checked. See our self-signed guide for details.
- **max_connections** – Maximum allowed number of simultaneous HTTPS connections to the webhook for update delivery, 1-100. Defaults to 40. Use lower values to limit the load on your bot's server, and higher values to increase your bot's throughput.

- **allowed_updates** – A JSON-serialized list of the update types you want your bot to receive. For example, specify [“message”, “edited_channel_post”, “callback_query”] to only receive updates of these types. See Update for a complete list of available update types. Specify an empty list to receive all updates regardless of type (default). If not specified, the previous setting will be used.
- **ip_address** – The fixed IP address which will be used to send webhook requests instead of the IP address resolved through DNS
- **drop_pending_updates** – Pass True to drop all pending updates
- **timeout** – Integer. Request connection timeout
- **secret_token** – Secret token to be used to verify the webhook

Returns

setup_middleware(*middleware*)

Setup middleware.

Parameters
middleware – Middleware-class.

Returns

shipping_query_handler(*func*, ***kwargs*)

Shipping request handler.

Parameters

- **func** –
- **kwargs** –

Returns

async skip_updates()

Skip existing updates. Only last update will remain on server.

async stop_message_live_location(*chat_id: Optional[Union[int, str]] = None*, *message_id: Optional[int] = None*, *inline_message_id: Optional[str] = None*, *reply_markup: Optional[Union[*InlineKeyboardMarkup*,* ReplyKeyboardMarkup*,* ReplyKeyboardRemove*,* ForceReply*]] = None*, *timeout: Optional[int] = None*) → *Message*

Use this method to stop updating a live location message sent by the bot or via the bot (for inline bots) before live_period expires.

Telegram documentation: https://core.telegram.org/bots/api#stopmessagelivelocation

Parameters

- **chat_id** –
- **message_id** –
- **inline_message_id** –
- **reply_markup** –
- **timeout** –

Returns

async stop_poll(*chat_id: Union[int, str]*, *message_id: int*, *reply_markup: Optional[Union[*InlineKeyboardMarkup, ReplyKeyboardMarkup, ReplyKeyboardRemove, ForceReply*]] = None*) → *Poll*

Stops poll.

Telegram documentation: https://core.telegram.org/bots/api#stoppoll

Parameters

- **chat_id** –
- **message_id** –
- **reply_markup** –

Returns

async unban_chat_member(*chat_id: Union[int, str]*, *user_id: int*, *only_if_banned: Optional[bool] = False*) → bool

Use this method to unban a previously kicked user in a supergroup or channel. The user will not return to the group or channel automatically, but will be able to join via link, etc. The bot must be an administrator for this to work. By async default, this method guarantees that after the call the user is not a member of the chat, but will be able to join it. So if the user is a member of the chat they will also be removed from the chat. If you don't want this, use the parameter only_if_banned.

Telegram documentation: https://core.telegram.org/bots/api#unbanchatmember

Parameters

- **chat_id** – Unique identifier for the target group or username of the target supergroup or channel (in the format @username)
- **user_id** – Unique identifier of the target user
- **only_if_banned** – Do nothing if the user is not banned

Returns
True on success

async unban_chat_sender_chat(*chat_id: Union[int, str]*, *sender_chat_id: Union[int, str]*) → bool

Use this method to unban a previously banned channel chat in a supergroup or channel. The bot must be an administrator for this to work and must have the appropriate administrator rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#unbanchatsenderchat

Params
Parameters

- **chat_id** – Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **sender_chat_id** – Unique identifier of the target sender chat

Returns
True on success.

async unpin_all_chat_messages(*chat_id: Union[int, str]*) → bool

Use this method to unpin a all pinned messages in a supergroup chat. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#unpinallchatmessages

Parameters
chat_id – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)

Returns

async unpin_chat_message(*chat_id: Union[int, str]*, *message_id: Optional[int] = None*) → bool

Use this method to unpin specific pinned message in a supergroup chat. The bot must be an administrator in the chat for this to work and must have the appropriate admin rights. Returns True on success.

Telegram documentation: https://core.telegram.org/bots/api#unpinchatmessage

Parameters

- **chat_id** – Int or Str: Unique identifier for the target chat or username of the target channel (in the format @channelusername)
- **message_id** – Int: Identifier of a message to unpin

Returns

async upload_sticker_file(*user_id: int*, *png_sticker: Union[Any, str]*) → *File*

Use this method to upload a .png file with a sticker for later use in createNewStickerSet and addStickerToSet methods (can be used multiple times). Returns the uploaded File on success.

Telegram documentation: https://core.telegram.org/bots/api#uploadstickerfile

Parameters

- **user_id** –
- **png_sticker** –

Returns

class telebot.async_telebot.**ExceptionHandler**

Bases: object

Class for handling exceptions while Polling

handle(*exception*)

class telebot.async_telebot.**Handler**(*callback*, **args*, ***kwargs*)

Bases: object

Class for (next step|reply) handlers

telebot.async_telebot.**REPLY_MARKUP_TYPES**

telebot

Type
Module

alias of Union[*InlineKeyboardMarkup*, *ReplyKeyboardMarkup*, *ReplyKeyboardRemove*, *ForceReply*]

Asyncio filters

class telebot.asyncio_filters.**AdvancedCustomFilter**

Bases: ABC

Simple Custom Filter base class. Create child class with check() method. Accepts two parameters, returns bool: True - filter passed, False - filter failed. message: Message class text: Filter value given in handler

Child classes should have .key property.

async check(*message*, *text*)

Perform a check.

key: str = None

class telebot.asyncio_filters.**ChatFilter**

Bases: *AdvancedCustomFilter*

Check whether chat_id corresponds to given chat_id.

Example: @bot.message_handler(chat_id=[99999])

async check(*message*, *text*)

Perform a check.

key: str = 'chat_id'

class telebot.asyncio_filters.**ForwardFilter**

Bases: *SimpleCustomFilter*

Check whether message was forwarded from channel or group.

Example:

@bot.message_handler(is_forwarded=True)

async check(*message*)

Perform a check.

key: str = 'is_forwarded'

class telebot.asyncio_filters.**IsAdminFilter**(*bot*)

Bases: *SimpleCustomFilter*

Check whether the user is administrator / owner of the chat.

Example: @bot.message_handler(chat_types=['supergroup'], is_chat_admin=True)

async check(*message*)

Perform a check.

key: str = 'is_chat_admin'

class telebot.asyncio_filters.**IsDigitFilter**

Bases: *SimpleCustomFilter*

Filter to check whether the string is made up of only digits.

Example: @bot.message_handler(is_digit=True)

async check(*message*)

Perform a check.

key: str = 'is_digit'

class telebot.asyncio_filters.**IsReplyFilter**

Bases: *SimpleCustomFilter*

Check whether message is a reply.

Example:

@bot.message_handler(is_reply=True)

async check(*message*)

Perform a check.

key: str = 'is_reply'

class telebot.asyncio_filters.**LanguageFilter**

Bases: *AdvancedCustomFilter*

Check users language_code.

Example:

@bot.message_handler(language_code=['ru'])

async check(*message*, *text*)

Perform a check.

key: str = 'language_code'

class telebot.asyncio_filters.**SimpleCustomFilter**

Bases: ABC

Simple Custom Filter base class. Create child class with check() method. Accepts only message, returns bool value, that is compared with given in handler.

Child classes should have .key property.

async check(*message*)

Perform a check.

key: str = None

class telebot.asyncio_filters.**StateFilter**(*bot*)

Bases: *AdvancedCustomFilter*

Filter to check state.

Example: @bot.message_handler(state=1)

async check(*message*, *text*)

Perform a check.

key: str = 'state'

class telebot.asyncio_filters.**TextContainsFilter**

Bases: *AdvancedCustomFilter*

Filter to check Text message. key: text

Example: # Will respond if any message.text contains word 'account' @bot.message_handler(text_contains=['account'])

async check(*message*, *text*)

Perform a check.

key: str = 'text_contains'

class telebot.asyncio_filters.TextFilter(*equals: Optional[str] = None*, *contains: Optional[Union[list, tuple]] = None*, *starts_with: Optional[Union[str, list, tuple]] = None*, *ends_with: Optional[Union[str, list, tuple]] = None*, *ignore_case: bool = False*)

Bases: object

Advanced text filter to check (types.Message, types.CallbackQuery, types.InlineQuery, types.Poll)

example of usage is in examples/asynchronous_telebot/custom_filters/advanced_text_filter.py

async check(*obj: Union[*Message, CallbackQuery, InlineQuery, Poll*]*)

class telebot.asyncio_filters.TextMatchFilter

Bases: *AdvancedCustomFilter*

Filter to check Text message. key: text

Example: @bot.message_handler(text=['account'])

async check(*message*, *text*)

Perform a check.

key: str = 'text'

class telebot.asyncio_filters.TextStartsFilter

Bases: *AdvancedCustomFilter*

Filter to check whether message starts with some text.

Example: # Will work if message.text starts with 'Sir'. @bot.message_handler(text_startswith='Sir')

async check(*message*, *text*)

Perform a check.

key: str = 'text_startswith'

Asynchronous storage for states

class telebot.asyncio_storage.StateContext(*obj*, *chat_id*, *user_id*)

Bases: object

Class for data.

class telebot.asyncio_storage.StateMemoryStorage

Bases: *StateStorageBase*

async delete_state(*chat_id*, *user_id*)

Delete state for a particular user.

async get_data(*chat_id*, *user_id*)

Get data for a user in a particular chat.

get_interactive_data(*chat_id*, *user_id*)

async get_state(*chat_id*, *user_id*)

async reset_data(*chat_id*, *user_id*)

Reset data for a particular user in a chat.

async save(*chat_id*, *user_id*, *data*)

async set_data(*chat_id*, *user_id*, *key*, *value*)

Set data for a user in a particular chat.

async set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user.

! Note that you should create a record if it does not exist, and if a record with state already exists, you need to update a record.

class telebot.asyncio_storage.StatePickleStorage(*file_path='./.state-save/states.pkl'*)

Bases: *StateStorageBase*

async convert_old_to_new()

create_dir()

Create directory .save-handlers.

async delete_state(*chat_id*, *user_id*)

Delete state for a particular user.

async get_data(*chat_id*, *user_id*)

Get data for a user in a particular chat.

get_interactive_data(*chat_id*, *user_id*)

async get_state(*chat_id*, *user_id*)

read()

async reset_data(*chat_id*, *user_id*)

Reset data for a particular user in a chat.

async save(*chat_id*, *user_id*, *data*)

async set_data(*chat_id*, *user_id*, *key*, *value*)

Set data for a user in a particular chat.

async set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user.

! Note that you should create a record if it does not exist, and if a record with state already exists, you need to update a record.

update_data()

class telebot.asyncio_storage.StateRedisStorage(*host='localhost'*, *port=6379*, *db=0*, *password=None*, *prefix='telebot_'*)

Bases: *StateStorageBase*

This class is for Redis storage. This will work only for states. To use it, just pass this class to: TeleBot(storage=StateRedisStorage())

async delete_record(*key*)

Function to delete record from database. It has nothing to do with states. Made for backward compatibility

async delete_state(*chat_id*, *user_id*)

Delete state for a particular user in a chat.

async get_data(*chat_id*, *user_id*)

Get data of particular user in a particular chat.

get_interactive_data(*chat_id*, *user_id*)

Get Data in interactive way. You can use with() with this function.

async get_record(*key*)

Function to get record from database. It has nothing to do with states. Made for backward compatibility

async get_state(*chat_id*, *user_id*)

Get state of a user in a chat.

async get_value(*chat_id*, *user_id*, *key*)

Get value for a data of a user in a chat.

async reset_data(*chat_id*, *user_id*)

Reset data of a user in a chat.

async save(*chat_id*, *user_id*, *data*)

async set_data(*chat_id*, *user_id*, *key*, *value*)

Set data without interactive data.

async set_record(*key*, *value*)

Function to set record to database. It has nothing to do with states. Made for backward compatibility

async set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user in a chat.

class telebot.asyncio_storage.StateStorageBase

Bases: object

async delete_state(*chat_id*, *user_id*)

Delete state for a particular user.

async get_data(*chat_id*, *user_id*)

Get data for a user in a particular chat.

async get_state(*chat_id*, *user_id*)

async reset_data(*chat_id*, *user_id*)

Reset data for a particular user in a chat.

async save(*chat_id*, *user_id*, *data*)

async set_data(*chat_id*, *user_id*, *key*, *value*)

Set data for a user in a particular chat.

async set_state(*chat_id*, *user_id*, *state*)

Set state for a particular user.

! Note that you should create a record if it does not exist, and if a record with state already exists, you need to update a record.

Asyncio handler backends

class telebot.asyncio_handler_backends.**BaseMiddleware**

Bases: object

Base class for middleware. Your middlewares should be inherited from this class.

async **post_process**(*message*, *data*, *exception*)

async **pre_process**(*message*, *data*)

class telebot.asyncio_handler_backends.**CancelUpdate**

Bases: object

Class for canceling updates. Just return instance of this class in middleware to skip update. Update will skip handler and execution of post_process in middlewares.

class telebot.asyncio_handler_backends.**SkipHandler**

Bases: object

Class for skipping handlers. Just return instance of this class in middleware to skip handler. Update will go to post_process, but will skip execution of handler.

class telebot.asyncio_handler_backends.**State**

Bases: object

class telebot.asyncio_handler_backends.**StatesGroup**

Bases: object

1.3.6 Callback data factory

callback_data file

This file was added during the pull request. The maintainers overlooked that it was copied "as is" from another project and they do not consider it as a right way to develop a project. However, due to backward compatibility we had to leave this file in the project with the above copyright added, as it is required by the original project license.

class telebot.callback_data.**CallbackData**(**parts*, *prefix: str*, *sep=':'*)

Bases: object

Callback data factory This class will help you to work with CallbackQuery

filter(**config*) → *CallbackDataFilter*

Generate filter

Parameters
config – specified named parameters will be checked with CallbackQury.data

Returns
CallbackDataFilter class

new(**args*, ***kwargs*) → str

Generate callback data

Parameters

- **args** – positional parameters of CallbackData instance parts
- **kwargs** – named parameters

Returns
str

parse(*callback_data: str*) → Dict[str, str]

Parse data from the callback data

Parameters
callback_data – string, use to telebot.types.CallbackQuery to parse it from string to a dict

Returns
dict parsed from callback data

class telebot.callback_data.**CallbackDataFilter**(*factory*, *config: Dict[str, str]*)

Bases: object

check(*query*)

Checks if query.data appropriates to specified config

Parameters
query – telebot.types.CallbackQuery

Returns
bool

1.3.7 Utils

util file

class telebot.util.**AsyncTask**(*target*, **args*, ***kwargs*)

Bases: object

wait()

class telebot.util.**CustomRequestResponse**(*json_text*, *status_code=200*, *reason=''*)

Bases: object

json()

telebot.util.**OrEvent**(**events*)

class telebot.util.**ThreadPool**(*telebot*, *num_threads=2*)

Bases: object

clear_exceptions()

close()

on_exception(*worker_thread*, *exc_info*)

put(*func*, **args*, ***kwargs*)

raise_exceptions()

class telebot.util.**WorkerThread**(*exception_callback=None*, *queue=None*, *name=None*)

Bases: Thread

clear_exceptions()

count = 0

put(*task*, **args*, ***kwargs*)

raise_exceptions()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()

telebot.util.**antiflood**(*function*, **args*, ***kwargs*)

Use this function inside loops in order to avoid getting TooManyRequests error. Example:

```
from telebot.util import antiflood
for chat_id in chat_id_list:
msg = antiflood(bot.send_message, chat_id, text)
```

Parameters

- **function** –
- **args** –
- **kwargs** –

Returns

None

telebot.util.**async_dec**()

telebot.util.**chunks**(*lst*, *n*)

Yield successive n-sized chunks from lst.

telebot.util.**deprecated**(*warn: bool = True*, *alternative: Optional[Callable] = None*, *deprecation_text=None*)

Use this decorator to mark functions as deprecated. When the function is used, an info (or warning if *warn* is True) is logged.

Parameters

- **warn** – If True a warning is logged else an info
- **alternative** – The new function to use instead
- **deprecation_text** – Custom deprecation text

telebot.util.**escape**(*text: str*) → str

Replaces the following chars in *text* ('&' with '&', '<' with '<' and '>' with '>').

Parameters
text – the text to escape

Returns
the escaped text

telebot.util.**extract_arguments**(*text: str*) → str

Returns the argument after the command.

Examples: extract_arguments("/get name"): 'name' extract_arguments("/get"): '' extract_arguments("/get@botName name"): 'name'

Parameters
text – String to extract the arguments from a command

Returns
the arguments if *text* is a command (according to is_command), else None.

telebot.util.**extract_command**(*text: str*) → Optional[str]

Extracts the command from *text* (minus the '/') if *text* is a command (see is_command). If *text* is not a command, this function returns None.

Examples: extract_command('/help'): 'help' extract_command('/help@BotName'): 'help' extract_command('/search black eyed peas'): 'search' extract_command('Good day to you'): None

Parameters
text – String to extract the command from

Returns
the command if *text* is a command (according to is_command), else None.

telebot.util.**generate_random_token**()

telebot.util.**is_bytes**(*var*)

telebot.util.**is_command**(*text: str*) → bool

Checks if *text* is a command. Telegram chat commands start with the '/' character.

Parameters
text – Text to check.

Returns
True if *text* is a command, else False.

telebot.util.**is_dict**(*var*)

telebot.util.**is_pil_image**(*var*)

telebot.util.**is_string**(*var*)

telebot.util.**or_clear**(*self*)

telebot.util.**or_set**(*self*)

telebot.util.**orify**(*e*, *changed_callback*)

telebot.util.**parse_web_app_data**(*token: str*, *raw_init_data: str*)

telebot.util.**per_thread**(*key*, *construct_value*, *reset=False*)

telebot.util.**pil_image_to_file**(*image*, *extension='JPEG'*, *quality='web_low'*)

telebot.util.**quick_markup**(*values: Dict[str, Dict[str, Any]]*, *row_width: int = 2*) → *InlineKeyboardMarkup*

Returns a reply markup from a dict in this format: {'text': kwargs} This is useful to avoid always typing 'btn1 = InlineKeyboardButton(...)' 'btn2 = InlineKeyboardButton(...)'

Example:

```
quick_markup({
    'Twitter': {'url': 'https://twitter.com'},
    'Facebook': {'url': 'https://facebook.com'},
    'Back': {'callback_data': 'whatever'}
}, row_width=2):
    # returns an InlineKeyboardMarkup with two buttons in a row, one leading to␣
↪Twitter, the other to facebook
    # and a back button below

# kwargs can be:
{
    'url': None,
    'callback_data': None,
    'switch_inline_query': None,
    'switch_inline_query_current_chat': None,
    'callback_game': None,
    'pay': None,
    'login_url': None,
    'web_app': None
}
```

Parameters

- **values** – a dict containing all buttons to create in this format: {text: kwargs} {str:}
- **row_width** – int row width

Returns

InlineKeyboardMarkup

telebot.util.**smart_split**(*text: str*, *chars_per_string: int = 4096*) → List[str]

Splits one string into multiple strings, with a maximum amount of *chars_per_string* characters per string. This is very useful for splitting one giant message into multiples. If *chars_per_string* > 4096: *chars_per_string* = 4096. Splits by 'n', '. ' or ' ' in exactly this priority.

Parameters

- **text** – The text to split

- **chars_per_string** – The number of maximum characters per part the text is split to.

Returns
The splitted text as a list of strings.

telebot.util.**split_string**(*text: str*, *chars_per_string: int*) → List[str]

Splits one string into multiple strings, with a maximum amount of *chars_per_string* characters per string. This is very useful for splitting one giant message into multiples.

Parameters

- **text** – The text to split
- **chars_per_string** – The number of characters per line the text is split into.

Returns
The splitted text as a list of strings.

telebot.util.**user_link**(*user:* User, *include_id: bool = False*) → str

Returns an HTML user link. This is useful for reports. Attention: Don't forget to set parse_mode to 'HTML'!

Example: bot.send_message(your_user_id, user_link(message.from_user) + ' started the bot!', parse_mode='HTML')

Parameters

- **user** – the user (not the user_id)
- **include_id** – include the user_id

Returns
HTML user link

telebot.util.**validate_web_app_data**(*token*, *raw_init_data*)

telebot.util.**webhook_google_functions**(*bot*, *request*)

A webhook endpoint for Google Cloud Functions FaaS.

1.3.8 Formatting options

Markdown & HTML formatting functions.

New in version 4.5.1.

telebot.formatting.**escape_html**(*content: str*) → str

Escapes HTML characters in a string of HTML.

Parameters
content – The string of HTML to escape.

telebot.formatting.**escape_markdown**(*content: str*) → str

Escapes Markdown characters in a string of Markdown.

Credits: simonsmh

Parameters
content – The string of Markdown to escape.

telebot.formatting.**format_text**(**args*, *separator='\n'*)

Formats a list of strings into a single string.

```
format_text( # just an example
    mbold('Hello'),
    mitalic('World')
)
```

Parameters

- **args** – Strings to format.
- **separator** – The separator to use between each string.

telebot.formatting.**hbold**(*content: str*, *escape: bool = True*) → str

Returns an HTML-formatted bold string.

Parameters

- **content** – The string to bold.
- **escape** – True if you need to escape special characters.

telebot.formatting.**hcode**(*content: str*, *escape: bool = True*) → str

Returns an HTML-formatted code string.

Parameters

- **content** – The string to code.
- **escape** – True if you need to escape special characters.

telebot.formatting.**hide_link**(*url: str*) → str

Hide url of an image.

Parameters
url –

Returns
str

telebot.formatting.**hitalic**(*content: str*, *escape: bool = True*) → str

Returns an HTML-formatted italic string.

Parameters

- **content** – The string to italicize.
- **escape** – True if you need to escape special characters.

telebot.formatting.**hlink**(*content: str*, *url: str*, *escape: bool = True*) → str

Returns an HTML-formatted link string.

Parameters

- **content** – The string to link.
- **url** – The URL to link to.
- **escape** – True if you need to escape special characters.

telebot.formatting.**hpre**(*content: str*, *escape: bool = True*, *language: str = ''*) → str

Returns an HTML-formatted preformatted string.

Parameters

- **content** – The string to preformatted.
- **escape** – True if you need to escape special characters.

telebot.formatting.**hspoiler**(*content: str*, *escape: bool = True*) → str

Returns an HTML-formatted spoiler string.

Parameters

- **content** – The string to spoiler.
- **escape** – True if you need to escape special characters.

telebot.formatting.**hstrikethrough**(*content: str*, *escape: bool = True*) → str

Returns an HTML-formatted strikethrough string.

Parameters

- **content** – The string to strikethrough.
- **escape** – True if you need to escape special characters.

telebot.formatting.**hunderline**(*content: str*, *escape: bool = True*) → str

Returns an HTML-formatted underline string.

Parameters

- **content** – The string to underline.
- **escape** – True if you need to escape special characters.

telebot.formatting.**mbold**(*content: str*, *escape: bool = True*) → str

Returns a Markdown-formatted bold string.

Parameters

- **content** – The string to bold.
- **escape** – True if you need to escape special characters.

telebot.formatting.**mcode**(*content: str*, *language: str = ''*, *escape: bool = True*) → str

Returns a Markdown-formatted code string.

Parameters

- **content** – The string to code.
- **escape** – True if you need to escape special characters.

telebot.formatting.**mitalic**(*content: str*, *escape: bool = True*) → str

Returns a Markdown-formatted italic string.

Parameters

- **content** – The string to italicize.
- **escape** – True if you need to escape special characters.

telebot.formatting.**mlink**(*content: str*, *url: str*, *escape: bool = True*) → str

Returns a Markdown-formatted link string.

Parameters

- **content** – The string to link.
- **url** – The URL to link to.

- **escape** – True if you need to escape special characters.

telebot.formatting.**mspoiler**(*content: str*, *escape: bool = True*) → str

Returns a Markdown-formatted spoiler string.

Parameters

- **content** – The string to spoiler.
- **escape** – True if you need to escape special characters.

telebot.formatting.**mstrikethrough**(*content: str*, *escape: bool = True*) → str

Returns a Markdown-formatted strikethrough string.

Parameters

- **content** – The string to strikethrough.
- **escape** – True if you need to escape special characters.

telebot.formatting.**munderline**(*content: str*, *escape: bool = True*) → str

Returns a Markdown-formatted underline string.

Parameters

- **content** – The string to underline.
- **escape** – True if you need to escape special characters.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W