Московский государственный технический
университет им. Н.Э. Баумана.

Факультет «Информатика и системы управления»

Кафедра ИУ5. Курс «Базовые компоненты интернет-технологий»
Отчёт по лабораторной работе № 6.

Выполнил:
студент группы ИУ5-51б
Афанасьев Д. М.

Проверил:
преподаватель каф. ИУ5
Гапанюк Ю.Е.

Подпись и дата:

Подпись и дата:

Москва, 2022 г.

Задание

Разработайте простого бота для Telegram. Бот должен использовать функциональность создания кнопок.

Текст программы

main.py

```
import unittest
import telebot
import requests
from telebot import types

bot = telebot.TeleBot('5957234710:AAFt1IZ7KG4M37kxsbZFHOKTubOT2rk6YVo')
appid = '78db3afcc8f32dd3faf7e3ef49dd46a1'
s_city = "Moscow (RU)"
city_id = 0


@bot.message_handler(commands=['start'])
def start(message):
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True)
    btn1 = types.KeyboardButton("Погода")
    btn2 = types.KeyboardButton("Температура")
    markup.add(btn1, btn2)
    if type(message) != str:
        bot.send_message(message.chat.id, text="Я родился!",
reply_markup=markup)
    else:
        raise Exception("Message is str, not telebot object")

    global message1

    message1 = message

    if message == None:
        return None

    return message1


@bot.message_handler(content_types=['text'])
def func(message):
    try:
        res = requests.get(

f"http://api.openweathermap.org/data/2.5/forecast?id=524901&appid={appid}&lan
g=ru&units=metric")
        data = res.json()
        if (not data) or (str(data['cod']) == '404'):
            raise Exception('Page not Found 404')
    except Exception as e:
        bot.send_message(message.chat.id, text=f"Сервер упал :(\nОшибка:
{e}")
        return
    if message == None:
        return
    if message.text == "Погода":
        weather = data['list'][0]['weather'][0]['description'].title()
        bot.send_message(message.chat.id, text=weather)
        markup = types.ReplyKeyboardMarkup(resize_keyboard=True)
        back = types.KeyboardButton("Назад")
```

```
        markup.add(back)

    elif message.text == "Температура":
        temp = 'Температура ' + str(data['list'][0]['main']['temp']) + '°C'
        feels_temp = 'Ощущается как ' +
str(data['list'][0]['main']['feels_like']) + '°C'
        markup = types.ReplyKeyboardMarkup(resize_keyboard=True)
        bot.send_message(message.chat.id, text=temp + "\n" + feels_temp,
reply_markup=markup)
        back = types.KeyboardButton("Назад")
        markup.add(back)

    elif message.text == "Назад":
        markup = types.ReplyKeyboardMarkup(resize_keyboard=True)
        button1 = types.KeyboardButton("Погода")
        button2 = types.KeyboardButton("Температура")
        markup.add(button1, button2)
        bot.send_message(message.chat.id, text="Вы вернулись в главное меню,
чем могу помочь?", reply_markup=markup)
    else:
        bot.send_message(message.chat.id, text="Не знаю такого..")


class test_class(unittest.TestCase):
    def test_false_parameters(self):
        with self.assertRaises(Exception) as context:
            start("text")
        self.assertEqual(
            "Message is str, not telebot object",
            str(context.exception)
        )

    def test_zero_parameters(self):
        with self.assertRaises(TypeError) as context:
            start()
        self.assertEqual(
            "start() missing 1 required positional argument: 'message'",
            str(context.exception)
        )

    def test_func(self):
        message1 = None
        self.assertEqual(func(message1), None)

    def test_func_zero_parameters(self):
        with self.assertRaises(TypeError) as context:
            func()
        self.assertEqual(
            "func() missing 1 required positional argument: 'message'",
            str(context.exception)
        )


bot.polling(none_stop=True)
unittest.main()
```

util.py

```
# -*- coding: utf-8 -*-
import random
import re
import string
```

```
import threading
import traceback
from typing import Any, Callable, List, Dict, Optional, Union
import hmac
from hashlib import sha256
from urllib.parse import parse_qsl

# noinspection PyPep8Naming
import queue as Queue
import logging

from telebot import types

try:
    import ujson as json
except ImportError:
    import json

try:
    # noinspection PyPackageRequirements
    from PIL import Image
    from io import BytesIO

    pil_imported = True
except:
    pil_imported = False

MAX_MESSAGE_LENGTH = 4096

logger = logging.getLogger('TeleBot')

thread_local = threading.local()

#: Contains all media content types.
content_type_media = [
    'text', 'audio', 'document', 'animation', 'game', 'photo', 'sticker',
'video', 'video_note', 'voice', 'contact',
    'location', 'venue', 'dice', 'invoice', 'successful_payment',
'connected_website', 'poll', 'passport_data',
    'web_app_data',
]

#: Contains all service content types such as `User joined the group`.
content_type_service = [
    'new_chat_members', 'left_chat_member', 'new_chat_title',
'new_chat_photo', 'delete_chat_photo', 'group_chat_created',
    'supergroup_chat_created', 'channel_chat_created', 'migrate_to_chat_id',
'migrate_from_chat_id', 'pinned_message',
    'proximity_alert_triggered', 'video_chat_scheduled',
'video_chat_started', 'video_chat_ended',
    'video_chat_participants_invited', 'message_auto_delete_timer_changed',
'forum_topic_created', 'forum_topic_closed',
    'forum_topic_reopened',
]

#: All update types, should be used for allowed updates parameter in polling.
update_types = [
    "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",
]
```

```
class WorkerThread(threading.Thread):
    """
    :meta private:
    """
    count = 0

    def __init__(self, exception_callback=None, queue=None, name=None):
        if not name:
            name = "WorkerThread{0}".format(self.__class__.count + 1)
            self.__class__.count += 1
        if not queue:
            queue = Queue.Queue()

        threading.Thread.__init__(self, name=name)
        self.queue = queue
        self.daemon = True

        self.received_task_event = threading.Event()
        self.done_event = threading.Event()
        self.exception_event = threading.Event()
        self.continue_event = threading.Event()

        self.exception_callback = exception_callback
        self.exception_info = None
        self._running = True
        self.start()

    def run(self):
        while self._running:
            try:
                task, args, kwargs = self.queue.get(block=True, timeout=.5)
                self.continue_event.clear()
                self.received_task_event.clear()
                self.done_event.clear()
                self.exception_event.clear()
                logger.debug("Received task")
                self.received_task_event.set()

                task(*args, **kwargs)
                logger.debug("Task complete")
                self.done_event.set()
            except Queue.Empty:
                pass
            except Exception as e:
                logger.debug(type(e).__name__ + " occurred, args=" +
str(e.args) + "\n" + traceback.format_exc())
                self.exception_info = e
                self.exception_event.set()
                if self.exception_callback:
                    self.exception_callback(self, self.exception_info)
                self.continue_event.wait()

    def put(self, task, *args, **kwargs):
        self.queue.put((task, args, kwargs))

    def raise_exceptions(self):
        if self.exception_event.is_set():
            raise self.exception_info

    def clear_exceptions(self):
        self.exception_event.clear()
        self.continue_event.set()
```

```
    def stop(self):
        self._running = False


class ThreadPool:
    """
    :meta private:
    """
    def __init__(self, telebot, num_threads=2):
        self.telebot = telebot
        self.tasks = Queue.Queue()
        self.workers = [WorkerThread(self.on_exception, self.tasks) for _ in
range(num_threads)]
        self.num_threads = num_threads

        self.exception_event = threading.Event()
        self.exception_info = None

    def put(self, func, *args, **kwargs):
        self.tasks.put((func, args, kwargs))

    def on_exception(self, worker_thread, exc_info):
        if self.telebot.exception_handler is not None:
            handled = self.telebot.exception_handler.handle(exc_info)
        else:
            handled = False
        if not handled:
            self.exception_info = exc_info
            self.exception_event.set()
        worker_thread.continue_event.set()

    def raise_exceptions(self):
        if self.exception_event.is_set():
            raise self.exception_info

    def clear_exceptions(self):
        self.exception_event.clear()

    def close(self):
        for worker in self.workers:
            worker.stop()
        for worker in self.workers:
            worker.join()


class AsyncTask:
    """
    :meta private:
    """
    def __init__(self, target, *args, **kwargs):
        self.target = target
        self.args = args
        self.kwargs = kwargs

        self.done = False
        self.thread = threading.Thread(target=self._run)
        self.thread.start()

    def _run(self):
        try:
            self.result = self.target(*self.args, **self.kwargs)
        except Exception as e:
            self.result = e
        self.done = True
```

```
    def wait(self):
        if not self.done:
            self.thread.join()
        if isinstance(self.result, BaseException):
            raise self.result
        else:
            return self.result


class CustomRequestResponse():
    """
    :meta private:
    """
    def __init__(self, json_text, status_code = 200, reason = ""):
        self.status_code = status_code
        self.text = json_text
        self.reason = reason

    def json(self):
        return json.loads(self.text)


def async_dec():
    """
    :meta private:
    """
    def decorator(fn):
        def wrapper(*args, **kwargs):
            return AsyncTask(fn, *args, **kwargs)

        return wrapper

    return decorator


def is_string(var) -> bool:
    """
    Returns True if the given object is a string.
    """
    return isinstance(var, str)


def is_dict(var) -> bool:
    """
    Returns True if the given object is a dictionary.

    :param var: object to be checked
    :type var: :obj:`object`

    :return: True if the given object is a dictionary.
    :rtype: :obj:`bool`
    """
    return isinstance(var, dict)


def is_bytes(var) -> bool:
    """
    Returns True if the given object is a bytes object.

    :param var: object to be checked
    :type var: :obj:`object`

    :return: True if the given object is a bytes object.
```

```
    :rtype: :obj:`bool`
    """
    return isinstance(var, bytes)


def is_pil_image(var) -> bool:
    """
    Returns True if the given object is a PIL.Image.Image object.

    :param var: object to be checked
    :type var: :obj:`object`

    :return: True if the given object is a PIL.Image.Image object.
    :rtype: :obj:`bool`
    """
    return pil_imported and isinstance(var, Image.Image)


def pil_image_to_file(image, extension='JPEG', quality='web_low'):
    if pil_imported:
        photoBuffer = BytesIO()
        image.convert('RGB').save(photoBuffer, extension, quality=quality)
        photoBuffer.seek(0)

        return photoBuffer
    else:
        raise RuntimeError('PIL module is not imported')


def is_command(text: str) -> bool:
    r"""
    Checks if `text` is a command. Telegram chat commands start with the '/'
character.

    :param text: Text to check.
    :type text: :obj:`str`

    :return: True if `text` is a command, else False.
    :rtype: :obj:`bool`
    """
    if text is None: return False
    return text.startswith('/')


def extract_command(text: str) -> Union[str, None]:
    """
    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.

    .. code-block:: python3
        :caption: Examples:

        extract_command('/help'): 'help'
        extract_command('/help@BotName'): 'help'
        extract_command('/search black eyed peas'): 'search'
        extract_command('Good day to you'): None

    :param text: String to extract the command from
    :type text: :obj:`str`

    :return: the command if `text` is a command (according to is_command),
else None.
    :rtype: :obj:`str` or :obj:`None`
```

```
    """
    if text is None: return None
    return text.split()[0].split('@')[0][1:] if is_command(text) else None


def extract_arguments(text: str) -> str or None:
    """
    Returns the argument after the command.

    .. code-block:: python3
        :caption: Examples:

        extract_arguments("/get name"): 'name'
        extract_arguments("/get"): ''
        extract_arguments("/get@botName name"): 'name'

    :param text: String to extract the arguments from a command
    :type text: :obj:`str`

    :return: the arguments if `text` is a command (according to is_command),
else None.
    :rtype: :obj:`str` or :obj:`None`
    """
    regexp = re.compile(r"/\w*(@\w*)*\s*([\s\S]*)", re.IGNORECASE)
    result = regexp.match(text)
    return result.group(2) if is_command(text) else None


def 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.

    :param text: The text to split
    :type text: :obj:`str`

    :param chars_per_string: The number of characters per line the text is
split into.
    :type chars_per_string: :obj:`int`

    :return: The splitted text as a list of strings.
    :rtype: :obj:`list` of :obj:`str`
    """
    return [text[i:i + chars_per_string] for i in range(0, len(text),
chars_per_string)]


def smart_split(text: str, chars_per_string: int=MAX_MESSAGE_LENGTH) ->
List[str]:
    r"""
    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.

    :param text: The text to split
    :type text: :obj:`str`

    :param chars_per_string: The number of maximum characters per part the
text is split to.
    :type chars_per_string: :obj:`int`
```

```
    :return: The splitted text as a list of strings.
    :rtype: :obj:`list` of :obj:`str`
    """

    def _text_before_last(substr: str) -> str:
        return substr.join(part.split(substr)[:-1]) + substr

    if chars_per_string > MAX_MESSAGE_LENGTH: chars_per_string =
MAX_MESSAGE_LENGTH

    parts = []
    while True:
        if len(text) < chars_per_string:
            parts.append(text)
            return parts

        part = text[:chars_per_string]

        if "\n" in part: part = _text_before_last("\n")
        elif ". " in part: part = _text_before_last(". ")
        elif " " in part: part = _text_before_last(" ")

        parts.append(part)
        text = text[len(part):]


def escape(text: str) -> str:
    """
    Replaces the following chars in `text` ('&' with '&amp;', '<' with '&lt;'
and '>' with '&gt;').

    :param text: the text to escape
    :return: the escaped text
    """
    chars = {"&": "&amp;", "<": "&lt;", ">": "&gt;"}
    for old, new in chars.items(): text = text.replace(old, new)
    return text


def user_link(user: types.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'!


    .. code-block:: python3
        :caption: Example:

        bot.send_message(your_user_id, user_link(message.from_user) + '
started the bot!', parse_mode='HTML')

    .. note::
        You can use formatting.* for all other formatting options(bold,
italic, links, and etc.)
        This method is kept for backward compatibility, and it is recommended
to use formatting.* for
        more options.

    :param user: the user (not the user_id)
    :type user: :obj:`telebot.types.User`

    :param include_id: include the user_id
    :type include_id: :obj:`bool`
```

```
    :return: HTML user link
    :rtype: :obj:`str`
    """
    name = escape(user.first_name)
    return (f"<a href='tg://user?id={user.id}'>{name}</a>"
        + (f" (<pre>{user.id}</pre>)" if include_id else ""))


def quick_markup(values: Dict[str, Dict[str, Any]], row_width: int=2) ->
types.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:

    .. code-block:: python3
        :caption: Using quick_markup:

        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
        }

    :param values: a dict containing all buttons to create in this format:
{text: kwargs} {str:}
    :type values: :obj:`dict`

    :param row_width: int row width
    :type row_width: :obj:`int`

    :return: InlineKeyboardMarkup
    :rtype: :obj:`types.InlineKeyboardMarkup`
    """
    markup = types.InlineKeyboardMarkup(row_width=row_width)
    buttons = [
        types.InlineKeyboardButton(text=text, **kwargs)
        for text, kwargs in values.items()
    ]
    markup.add(*buttons)
    return markup


# CREDITS TO http://stackoverflow.com/questions/12317940#answer-12320352
def or_set(self):
    """
    :meta private:
```

```
    """
    self._set()
    self.changed()


def or_clear(self):
    """
    :meta private:
    """
    self._clear()
    self.changed()


def orify(e, changed_callback):
    """
    :meta private:
    """
    if not hasattr(e, "_set"):
        e._set = e.set
    if not hasattr(e, "_clear"):
        e._clear = e.clear
    e.changed = changed_callback
    e.set = lambda: or_set(e)
    e.clear = lambda: or_clear(e)


def OrEvent(*events):
    """
    :meta private:
    """
    or_event = threading.Event()

    def changed():
        bools = [ev.is_set() for ev in events]
        if any(bools):
            or_event.set()
        else:
            or_event.clear()

    def busy_wait():
        while not or_event.is_set():
            # noinspection PyProtectedMember
            or_event._wait(3)

    for e in events:
        orify(e, changed)
    or_event._wait = or_event.wait
    or_event.wait = busy_wait
    changed()
    return or_event


def per_thread(key, construct_value, reset=False):
    """
    :meta private:
    """
    if reset or not hasattr(thread_local, key):
        value = construct_value()
        setattr(thread_local, key, value)

    return getattr(thread_local, key)


def chunks(lst, n):
```

```
    """Yield successive n-sized chunks from lst."""
    # https://stackoverflow.com/a/312464/9935473
    for i in range(0, len(lst), n):
        yield lst[i:i + n]


def generate_random_token() -> str:
    """
    Generates a random token consisting of letters and digits, 16 characters
long.

    :return: a random token
    :rtype: :obj:`str`
    """
    return ''.join(random.sample(string.ascii_letters, 16))


def 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.

    :meta private:

    :param warn: If True a warning is logged else an info
    :type warn: :obj:`bool`

    :param alternative: The new function to use instead
    :type alternative: :obj:`Callable`

    :param deprecation_text: Custom deprecation text
    :type deprecation_text: :obj:`str`

    :return: The decorated function
    """
    def decorator(function):
        def wrapper(*args, **kwargs):
            info = f"`{function.__name__}` is deprecated."
            if alternative:
                info += f" Use `{alternative.__name__}` instead"
            if deprecation_text:
                info += " " + deprecation_text
            if not warn:
                logger.info(info)
            else:
                logger.warning(info)
            return function(*args, **kwargs)
        return wrapper
    return decorator


# Cloud helpers
def webhook_google_functions(bot, request):
    """
    A webhook endpoint for Google Cloud Functions FaaS.

    :param bot: The bot instance
    :type bot: :obj:`telebot.TeleBot` or
:obj:`telebot.async_telebot.AsyncTeleBot`

    :param request: The request object
    :type request: :obj:`flask.Request`
```

```
    :return: The response object
    """
    if request.is_json:
        try:
            request_json = request.get_json()
            update = types.Update.de_json(request_json)
            bot.process_new_updates([update])
            return ''
        except Exception as e:
            print(e)
            return 'Bot FAIL', 400
    else:
        return 'Bot ON'


def antiflood(function: Callable, *args, **kwargs):
    """
    Use this function inside loops in order to avoid getting TooManyRequests
error.
    Example:

    .. code-block:: python3

        from telebot.util import antiflood
        for chat_id in chat_id_list:
        msg = antiflood(bot.send_message, chat_id, text)

    :param function: The function to call
    :type function: :obj:`Callable`

    :param args: The arguments to pass to the function
    :type args: :obj:`tuple`

    :param kwargs: The keyword arguments to pass to the function
    :type kwargs: :obj:`dict`

    :return: None
    """
    from telebot.apihelper import ApiTelegramException
    from time import sleep

    try:
        return function(*args, **kwargs)
    except ApiTelegramException as ex:
        if ex.error_code == 429:
            sleep(ex.result_json['parameters']['retry_after'])
            return function(*args, **kwargs)
        else:
            raise


def parse_web_app_data(token: str, raw_init_data: str):
    """
    Parses web app data.

    :param token: The bot token
    :type token: :obj:`str`

    :param raw_init_data: The raw init data
    :type raw_init_data: :obj:`str`

    :return: The parsed init data
    """
```

```
    is_valid = validate_web_app_data(token, raw_init_data)
    if not is_valid:
        return False

    result = {}
    for key, value in parse_qsl(raw_init_data):
        try:
            value = json.loads(value)
        except json.JSONDecodeError:
            result[key] = value
        else:
            result[key] = value
    return result


def validate_web_app_data(token: str, raw_init_data: str):
    """
    Validates web app data.

    :param token: The bot token
    :type token: :obj:`str`

    :param raw_init_data: The raw init data
    :type raw_init_data: :obj:`str`

    :return: The parsed init data
    """
    try:
        parsed_data = dict(parse_qsl(raw_init_data))
    except ValueError:
        return False
    if "hash" not in parsed_data:
        return False

    init_data_hash = parsed_data.pop('hash')
    data_check_string = "\n".join(f"{key}={value}" for key, value in
sorted(parsed_data.items()))
    secret_key = hmac.new(key=b"WebAppData", msg=token.encode(),
digestmod=sha256)

    return hmac.new(secret_key.digest(), data_check_string.encode(),
sha256).hexdigest() == init_data_hash
```

types.py

```
# -*- coding: utf-8 -*-

from io import IOBase
import logging
import os
from pathlib import Path
from typing import Dict, List, Optional, Union
from abc import ABC

try:
    import ujson as json
except ImportError:
    import json

from telebot import util
```

```
DISABLE_KEYLEN_ERROR = False

logger = logging.getLogger('TeleBot')


class JsonSerializable(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.

    """

    def to_json(self):
        """
        Returns a JSON string representation of this class.

        :meta private:

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


class Dictionaryable(object):
    """
    Subclasses of this class are guaranteed to be able to be converted to
dictionary.
    All subclasses of this class must override to_dict.

    """

    def to_dict(self):
        """
        Returns a DICT with class field values

        :meta private:

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


class JsonDeserializable(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.
    """

    @classmethod
    def de_json(cls, json_string):
        """
        Returns an instance of this class from the given json dict or string.

        :meta private:

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

```
    @staticmethod
    def 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)

        :meta private:

        :param json_type: input json or parsed dict
        :param dict_copy: if dict is passed and it is changed outside -
should be True!
        :return: Dictionary parsed from json or original dict
        """
        if util.is_dict(json_type):
            return json_type.copy() if dict_copy else json_type
        elif util.is_string(json_type):
            return json.loads(json_type)
        else:
            raise ValueError("json_type should be a json dict or string.")

    def __str__(self):
        d = {
            x: y.__dict__ if hasattr(y, '__dict__') else y
            for x, y in self.__dict__.items()
        }
        return str(d)


class Update(JsonDeserializable):
    """
    This object represents an incoming update.At most one of the optional
parameters can be present in any given update.

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

    :param update_id: The update's unique identifier. Update identifiers
start from a certain positive number and
        increase sequentially. This ID becomes especially handy if you're
using webhooks, since it allows you to ignore
        repeated updates or to restore the correct update sequence, should
they get out of order. If there are no new updates
        for at least a week, then identifier of the next update will be
chosen randomly instead of sequentially.
    :type update_id: :obj:`int`

    :param message: Optional. New incoming message of any kind - text, photo,
sticker, etc.
    :type message: :class:`telebot.types.Message`

    :param edited_message: Optional. New version of a message that is known
to the bot and was edited
    :type edited_message: :class:`telebot.types.Message`

    :param channel_post: Optional. New incoming channel post of any kind -
text, photo, sticker, etc.
    :type channel_post: :class:`telebot.types.Message`

    :param edited_channel_post: Optional. New version of a channel post that
is known to the bot and was edited
    :type edited_channel_post: :class:`telebot.types.Message`
```

```
    :param inline_query: Optional. New incoming inline query
    :type inline_query: :class:`telebot.types.InlineQuery`

    :param chosen_inline_result: Optional. The result of an inline query that
was chosen by a user and sent to their chat
        partner. Please see our documentation on the feedback collecting for
details on how to enable these updates for your
        bot.
    :type chosen_inline_result: :class:`telebot.types.ChosenInlineResult`

    :param callback_query: Optional. New incoming callback query
    :type callback_query: :class:`telebot.types.CallbackQuery`

    :param shipping_query: Optional. New incoming shipping query. Only for
invoices with flexible price
    :type shipping_query: :class:`telebot.types.ShippingQuery`

    :param pre_checkout_query: Optional. New incoming pre-checkout query.
Contains full information about
        checkout
    :type pre_checkout_query: :class:`telebot.types.PreCheckoutQuery`

    :param poll: Optional. New poll state. Bots receive only updates about
stopped polls and polls, which are sent by the
        bot
    :type poll: :class:`telebot.types.Poll`

    :param poll_answer: Optional. A user changed their answer in a non-
anonymous poll. Bots receive new votes only in
        polls that were sent by the bot itself.
    :type poll_answer: :class:`telebot.types.PollAnswer`

    :param my_chat_member: Optional. The bot's chat member status was updated
in a chat. For private chats, this update
        is received only when the bot is blocked or unblocked by the user.
    :type my_chat_member: :class:`telebot.types.ChatMemberUpdated`

    :param chat_member: Optional. A chat member's status was updated in a
chat. The bot must be an administrator in the
        chat and must explicitly specify “chat_member” in the list of
allowed_updates to receive these updates.
    :type chat_member: :class:`telebot.types.ChatMemberUpdated`

    :param chat_join_request: Optional. A request to join the chat has been
sent. The bot must have the
        can invite users administrator right in the chat to receive these
updates.
    :type chat_join_request: :class:`telebot.types.ChatJoinRequest`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Update`

    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        update_id = obj['update_id']
        message = Message.de_json(obj.get('message'))
        edited_message = Message.de_json(obj.get('edited_message'))
        channel_post = Message.de_json(obj.get('channel_post'))
        edited_channel_post = Message.de_json(obj.get('edited_channel_post'))
        inline_query = InlineQuery.de_json(obj.get('inline_query'))
        chosen_inline_result =
```

```
ChosenInlineResult.de_json(obj.get('chosen_inline_result'))
        callback_query = CallbackQuery.de_json(obj.get('callback_query'))
        shipping_query = ShippingQuery.de_json(obj.get('shipping_query'))
        pre_checkout_query =
PreCheckoutQuery.de_json(obj.get('pre_checkout_query'))
        poll = Poll.de_json(obj.get('poll'))
        poll_answer = PollAnswer.de_json(obj.get('poll_answer'))
        my_chat_member = ChatMemberUpdated.de_json(obj.get('my_chat_member'))
        chat_member = ChatMemberUpdated.de_json(obj.get('chat_member'))
        chat_join_request =
ChatJoinRequest.de_json(obj.get('chat_join_request'))
        return cls(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)

    def __init__(self, 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):
        self.update_id = update_id
        self.message = message
        self.edited_message = edited_message
        self.channel_post = channel_post
        self.edited_channel_post = edited_channel_post
        self.inline_query = inline_query
        self.chosen_inline_result = chosen_inline_result
        self.callback_query = callback_query
        self.shipping_query = shipping_query
        self.pre_checkout_query = pre_checkout_query
        self.poll = poll
        self.poll_answer = poll_answer
        self.my_chat_member = my_chat_member
        self.chat_member = chat_member
        self.chat_join_request = chat_join_request


class ChatMemberUpdated(JsonDeserializable):
    """
    This object represents changes in the status of a chat member.

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

    :param chat: Chat the user belongs to
    :type chat: :class:`telebot.types.Chat`

    :param from_user: Performer of the action, which resulted in the change
    :type from_user: :class:`telebot.types.User`

    :param date: Date the change was done in Unix time
    :type date: :obj:`int`

    :param old_chat_member: Previous information about the chat member
    :type old_chat_member: :class:`telebot.types.ChatMember`

    :param new_chat_member: New information about the chat member
    :type new_chat_member: :class:`telebot.types.ChatMember`

    :param invite_link: Optional. Chat invite link, which was used by the
user to join the chat; for joining by invite
        link events only.
```

```
    :type invite_link: :class:`telebot.types.ChatInviteLink`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatMemberUpdated`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        obj['chat'] = Chat.de_json(obj['chat'])
        obj['from_user'] = User.de_json(obj.pop('from'))
        obj['old_chat_member'] = ChatMember.de_json(obj['old_chat_member'])
        obj['new_chat_member'] = ChatMember.de_json(obj['new_chat_member'])
        obj['invite_link'] = ChatInviteLink.de_json(obj.get('invite_link'))
        return cls(**obj)

    def __init__(self, chat, from_user, date, old_chat_member,
new_chat_member, invite_link=None, **kwargs):
        self.chat: Chat = chat
        self.from_user: User = from_user
        self.date: int = date
        self.old_chat_member: ChatMember = old_chat_member
        self.new_chat_member: ChatMember = new_chat_member
        self.invite_link: Optional[ChatInviteLink] = invite_link

    @property
    def difference(self) -> 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]}


        :return: Dict of differences
        :rtype: Dict[str, List]
        """
        old: Dict = self.old_chat_member.__dict__
        new: Dict = self.new_chat_member.__dict__
        dif = {}
        for key in new:
            if key == 'user': continue
            if new[key] != old[key]:
                dif[key] = [old[key], new[key]]
        return dif


class ChatJoinRequest(JsonDeserializable):
    """
    Represents a join request sent to a chat.

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

    :param chat: Chat to which the request was sent
    :type chat: :class:`telebot.types.Chat`

    :param from: User that sent the join request
    :type from_user: :class:`telebot.types.User`

    :param date: Date the request was sent in Unix time
    :type date: :obj:`int`
```

```
    :param bio: Optional. Bio of the user.
    :type bio: :obj:`str`

    :param invite_link: Optional. Chat invite link that was used by the user
to send the join request
    :type invite_link: :class:`telebot.types.ChatInviteLink`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatJoinRequest`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        obj['chat'] = Chat.de_json(obj['chat'])
        obj['from_user'] = User.de_json(obj['from'])
        obj['invite_link'] = ChatInviteLink.de_json(obj.get('invite_link'))
        return cls(**obj)

    def __init__(self, chat, from_user, date, bio=None, invite_link=None,
**kwargs):
        self.chat = chat
        self.from_user = from_user
        self.date = date
        self.bio = bio
        self.invite_link = invite_link

class WebhookInfo(JsonDeserializable):
    """
    Describes the current status of a webhook.

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

    :param url: Webhook URL, may be empty if webhook is not set up
    :type url: :obj:`str`

    :param has_custom_certificate: True, if a custom certificate was provided
for webhook certificate checks
    :type has_custom_certificate: :obj:`bool`

    :param pending_update_count: Number of updates awaiting delivery
    :type pending_update_count: :obj:`int`

    :param ip_address: Optional. Currently used webhook IP address
    :type ip_address: :obj:`str`

    :param last_error_date: Optional. Unix time for the most recent error
that happened when trying to deliver an
        update via webhook
    :type last_error_date: :obj:`int`

    :param last_error_message: Optional. Error message in human-readable
format for the most recent error that
        happened when trying to deliver an update via webhook
    :type last_error_message: :obj:`str`

    :param last_synchronization_error_date: Optional. Unix time of the most
recent error that happened when trying
        to synchronize available updates with Telegram datacenters
    :type last_synchronization_error_date: :obj:`int`

    :param max_connections: Optional. The maximum allowed number of
simultaneous HTTPS connections to the webhook
        for update delivery
```

```
    :type max_connections: :obj:`int`

    :param allowed_updates: Optional. A list of update types the bot is
subscribed to. Defaults to all update types
        except chat_member
    :type allowed_updates: :obj:`list` of :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.WebhookInfo`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, 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):
        self.url = url
        self.has_custom_certificate = has_custom_certificate
        self.pending_update_count = pending_update_count
        self.ip_address = ip_address
        self.last_error_date = last_error_date
        self.last_error_message = last_error_message
        self.last_synchronization_error_date =
last_synchronization_error_date
        self.max_connections = max_connections
        self.allowed_updates = allowed_updates


class User(JsonDeserializable, Dictionaryable, JsonSerializable):
    """
    This object represents a Telegram user or bot.

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

    :param id: Unique identifier for this user or bot. This number may have
more than 32 significant bits and some
        programming languages may have difficulty/silent defects in
interpreting it. But it has at most 52 significant
        bits, so a 64-bit integer or double-precision float type are safe for
storing this identifier.
    :type id: :obj:`int`

    :param is_bot: True, if this user is a bot
    :type is_bot: :obj:`bool`

    :param first_name: User's or bot's first name
    :type first_name: :obj:`str`

    :param last_name: Optional. User's or bot's last name
    :type last_name: :obj:`str`

    :param username: Optional. User's or bot's username
    :type username: :obj:`str`

    :param language_code: Optional. IETF language tag of the user's language
    :type language_code: :obj:`str`

    :param is_premium: Optional. :obj:`bool`, if this user is a Telegram
Premium user
```

```
    :type is_premium: :obj:`bool`

    :param added_to_attachment_menu: Optional. :obj:`bool`, if this user
added the bot to the attachment menu
    :type added_to_attachment_menu: :obj:`bool`

    :param can_join_groups: Optional. True, if the bot can be invited to
groups. Returned only in getMe.
    :type can_join_groups: :obj:`bool`

    :param can_read_all_group_messages: Optional. True, if privacy mode is
disabled for the bot. Returned only in
        getMe.
    :type can_read_all_group_messages: :obj:`bool`

    :param supports_inline_queries: Optional. True, if the bot supports
inline queries. Returned only in getMe.
    :type supports_inline_queries: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.User`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, 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):
        self.id: int = id
        self.is_bot: bool = is_bot
        self.first_name: str = first_name
        self.username: str = username
        self.last_name: str = last_name
        self.language_code: str = language_code
        self.can_join_groups: bool = can_join_groups
        self.can_read_all_group_messages: bool = can_read_all_group_messages
        self.supports_inline_queries: bool = supports_inline_queries
        self.is_premium: bool = is_premium
        self.added_to_attachment_menu: bool = added_to_attachment_menu


    @property
    def full_name(self):
        """
        :return: User's full name
        """
        full_name = self.first_name
        if self.last_name:
            full_name += ' {0}'.format(self.last_name)
        return full_name

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {'id': self.id,
                'is_bot': self.is_bot,
                'first_name': self.first_name,
                'last_name': self.last_name,
```

```
                'username': self.username,
                'language_code': self.language_code,
                'can_join_groups': self.can_join_groups,
                'can_read_all_group_messages':
self.can_read_all_group_messages,
                'supports_inline_queries': self.supports_inline_queries,
                'is_premium': self.is_premium,
                'added_to_attachment_menu': self.added_to_attachment_menu}


class GroupChat(JsonDeserializable):
    """
    :meta private:
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, id, title, **kwargs):
        self.id: int = id
        self.title: str = title


class Chat(JsonDeserializable):
    """
    This object represents a chat.

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

    :param id: Unique identifier for this chat. This number may have more
than 32 significant bits and some programming
        languages may have difficulty/silent defects in interpreting it. But
it has at most 52 significant bits, so a signed
        64-bit integer or double-precision float type are safe for storing
this identifier.
    :type id: :obj:`int`

    :param type: Type of chat, can be either “private”, “group”, “supergroup”
or “channel”
    :type type: :obj:`str`

    :param title: Optional. Title, for supergroups, channels and group chats
    :type title: :obj:`str`

    :param username: Optional. Username, for private chats, supergroups and
channels if available
    :type username: :obj:`str`

    :param first_name: Optional. First name of the other party in a private
chat
    :type first_name: :obj:`str`

    :param last_name: Optional. Last name of the other party in a private
chat
    :type last_name: :obj:`str`

    :param is_forum: Optional. True, if the supergroup chat is a forum (has
topics enabled)
    :type is_forum: :obj:`bool`

    :param photo: Optional. Chat photo. Returned only in getChat.
    :type photo: :class:`telebot.types.ChatPhoto`
```

```
    :param active_usernames: Optional. If non-empty, the list of all active
chat usernames; for private chats, supergroups and channels.
        Returned only in getChat.
    :type active_usernames: :obj:`list` of :obj:`str`

    :param emoji_status_custom_emoji_id: Optional. Custom emoji identifier of
emoji status of the other party in a private chat.
        Returned only in getChat.
    :type emoji_status_custom_emoji_id: :obj:`str`

    :param bio: Optional. Bio of the other party in a private chat. Returned
only in getChat.
    :type bio: :obj:`str`

    :param has_private_forwards: Optional. :obj:`bool`, if privacy settings
of the other party in the private chat
        allows to use tg://user?id=<user_id> links only in chats with the
user. Returned only in getChat.
    :type has_private_forwards: :obj:`bool`

    :param has_restricted_voice_and_video_messages: Optional. True, if the
privacy settings of the other party restrict sending voice and video note
messages
        in the private chat. Returned only in getChat.
    :type :obj:`bool`

    :param join_to_send_messages: Optional. :obj:`bool`, if users need to
join the supergroup before they can send
        messages. Returned only in getChat.
    :type join_to_send_messages: :obj:`bool`

    :param join_by_request: Optional. :obj:`bool`, if all users directly
joining the supergroup need to be approved
        by supergroup administrators. Returned only in getChat.
    :type join_by_request: :obj:`bool`

    :param description: Optional. Description, for groups, supergroups and
channel chats. Returned only in getChat.
    :type description: :obj:`str`

    :param invite_link: Optional. Primary invite link, for groups,
supergroups and channel chats. Returned only in
        getChat.
    :type invite_link: :obj:`str`

    :param pinned_message: Optional. The most recent pinned message (by
sending date). Returned only in getChat.
    :type pinned_message: :class:`telebot.types.Message`

    :param permissions: Optional. Default chat member permissions, for groups
and supergroups. Returned only in
        getChat.
    :type permissions: :class:`telebot.types.ChatPermissions`

    :param slow_mode_delay: Optional. For supergroups, the minimum allowed
delay between consecutive messages sent
        by each unpriviledged user; in seconds. Returned only in getChat.
    :type slow_mode_delay: :obj:`int`

    :param message_auto_delete_time: Optional. The time after which all
messages sent to the chat will be
        automatically deleted; in seconds. Returned only in getChat.
    :type message_auto_delete_time: :obj:`int`
```

```
    :param has_protected_content: Optional. :obj:`bool`, if messages from the
chat can't be forwarded to other
        chats. Returned only in getChat.
    :type has_protected_content: :obj:`bool`

    :param sticker_set_name: Optional. For supergroups, name of group sticker
set. Returned only in getChat.
    :type sticker_set_name: :obj:`str`

    :param can_set_sticker_set: Optional. :obj:`bool`, if the bot can change
the group sticker set. Returned only in
        getChat.
    :type can_set_sticker_set: :obj:`bool`

    :param linked_chat_id: Optional. Unique identifier for the linked chat,
i.e. the discussion group identifier for
        a channel and vice versa; for supergroups and channel chats. This
identifier may be greater than 32 bits and some
        programming languages may have difficulty/silent defects in
interpreting it. But it is smaller than 52 bits, so a
        signed 64 bit integer or double-precision float type are safe for
storing this identifier. Returned only in getChat.
    :type linked_chat_id: :obj:`int`

    :param location: Optional. For supergroups, the location to which the
supergroup is connected. Returned only in
        getChat.
    :type location: :class:`telebot.types.ChatLocation`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Chat`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'photo' in obj:
            obj['photo'] = ChatPhoto.de_json(obj['photo'])
        if 'pinned_message' in obj:
            obj['pinned_message'] = Message.de_json(obj['pinned_message'])
        if 'permissions' in obj:
            obj['permissions'] = ChatPermissions.de_json(obj['permissions'])
        if 'location' in obj:
            obj['location'] = ChatLocation.de_json(obj['location'])
        return cls(**obj)

    def __init__(self, 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,
has_restricted_voice_and_video_messages=None,
                 is_forum=None, active_usernames=None,
emoji_status_custom_emoji_id=None, **kwargs):
        self.id: int = id
        self.type: str = type
        self.title: str = title
        self.username: str = username
```

```
        self.first_name: str = first_name
        self.last_name: str = last_name
        self.is_forum: bool = is_forum
        self.photo: ChatPhoto = photo
        self.bio: str = bio
        self.join_to_send_messages: bool = join_to_send_messages
        self.join_by_request: bool = join_by_request
        self.has_private_forwards: bool = has_private_forwards
        self.has_restricted_voice_and_video_messages: bool =
has_restricted_voice_and_video_messages
        self.description: str = description
        self.invite_link: str = invite_link
        self.pinned_message: Message = pinned_message
        self.permissions: ChatPermissions = permissions
        self.slow_mode_delay: int = slow_mode_delay
        self.message_auto_delete_time: int = message_auto_delete_time
        self.has_protected_content: bool = has_protected_content
        self.sticker_set_name: str = sticker_set_name
        self.can_set_sticker_set: bool = can_set_sticker_set
        self.linked_chat_id: int = linked_chat_id
        self.location: ChatLocation = location
        self.active_usernames: List[str] = active_usernames
        self.emoji_status_custom_emoji_id: str = emoji_status_custom_emoji_id


class MessageID(JsonDeserializable):
    """
    This object represents a unique message identifier.

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

    :param message_id: Unique message identifier
    :type message_id: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.MessageId`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, message_id, **kwargs):
        self.message_id = message_id


class WebAppData(JsonDeserializable, Dictionaryable):
    """
    Describes data sent from a Web App to the bot.

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

    :param data: The data. Be aware that a bad client can send arbitrary data
in this field.
    :type data: :obj:`str`

    :param button_text: Text of the web_app keyboard button from which the
Web App was opened. Be aware that a bad client
        can send arbitrary data in this field.
    :type button_text: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.WebAppData`
```

```
    """

    def __init__(self, data, button_text):
        self.data = data
        self.button_text = button_text
    def to_dict(self):
        return {'data': self.data, 'button_text': self.button_text}

    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        return cls(**obj)


class Message(JsonDeserializable):
    """
    This object represents a message.

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

    :param message_id: Unique message identifier inside this chat
    :type message_id: :obj:`int`

    :param message_thread_id: Optional. Unique identifier of a message thread
to which the message belongs; for supergroups only
    :type message_thread_id: :obj:`int`

    :param from_user: Optional. Sender of the message; empty for messages
sent to channels. For backward compatibility, the
        field contains a fake sender user in non-channel chats, if the
message was sent on behalf of a chat.
    :type from_user: :class:`telebot.types.User`

    :param sender_chat: Optional. Sender of the message, sent on behalf of a
chat. For example, the channel itself for
        channel posts, the supergroup itself for messages from anonymous
group administrators, the linked channel for
        messages automatically forwarded to the discussion group. For
backward compatibility, the field from contains a
        fake sender user in non-channel chats, if the message was sent on
behalf of a chat.
    :type sender_chat: :class:`telebot.types.Chat`

    :param date: Date the message was sent in Unix time
    :type date: :obj:`int`

    :param chat: Conversation the message belongs to
    :type chat: :class:`telebot.types.Chat`

    :param forward_from: Optional. For forwarded messages, sender of the
original message
    :type forward_from: :class:`telebot.types.User`

    :param forward_from_chat: Optional. For messages forwarded from channels
or from anonymous administrators,
        information about the original sender chat
    :type forward_from_chat: :class:`telebot.types.Chat`

    :param forward_from_message_id: Optional. For messages forwarded from
channels, identifier of the original
        message in the channel
    :type forward_from_message_id: :obj:`int`
```

```
    :param forward_signature: Optional. For forwarded messages that were
originally sent in channels or by an
        anonymous chat administrator, signature of the message sender if
present
    :type forward_signature: :obj:`str`

    :param forward_sender_name: Optional. Sender's name for messages
forwarded from users who disallow adding a link
        to their account in forwarded messages
    :type forward_sender_name: :obj:`str`

    :param forward_date: Optional. For forwarded messages, date the original
message was sent in Unix time
    :type forward_date: :obj:`int`

    :param is_topic_message: Optional. True, if the message is sent to a
forum topic
    :type is_topic_message: :obj:`bool`

    :param is_automatic_forward: Optional. :obj:`bool`, if the message is a
channel post that was automatically
        forwarded to the connected discussion group
    :type is_automatic_forward: :obj:`bool`

    :param reply_to_message: Optional. For replies, the original message.
Note that the Message object in this field
        will not contain further reply_to_message fields even if it itself is
a reply.
    :type reply_to_message: :class:`telebot.types.Message`

    :param via_bot: Optional. Bot through which the message was sent
    :type via_bot: :class:`telebot.types.User`

    :param edit_date: Optional. Date the message was last edited in Unix time
    :type edit_date: :obj:`int`

    :param has_protected_content: Optional. :obj:`bool`, if the message can't
be forwarded
    :type has_protected_content: :obj:`bool`

    :param media_group_id: Optional. The unique identifier of a media message
group this message belongs to
    :type media_group_id: :obj:`str`

    :param author_signature: Optional. Signature of the post author for
messages in channels, or the custom title of an
        anonymous group administrator
    :type author_signature: :obj:`str`

    :param text: Optional. For text messages, the actual UTF-8 text of the
message
    :type text: :obj:`str`

    :param entities: Optional. For text messages, special entities like
usernames, URLs, bot commands, etc. that
        appear in the text
    :type entities: :obj:`list` of :class:`telebot.types.MessageEntity`

    :param animation: Optional. Message is an animation, information about
the animation. For backward
        compatibility, when this field is set, the document field will also
be set
    :type animation: :class:`telebot.types.Animation`
```

```
    :param audio: Optional. Message is an audio file, information about the
file
    :type audio: :class:`telebot.types.Audio`

    :param document: Optional. Message is a general file, information about
the file
    :type document: :class:`telebot.types.Document`

    :param photo: Optional. Message is a photo, available sizes of the photo
    :type photo: :obj:`list` of :class:`telebot.types.PhotoSize`

    :param sticker: Optional. Message is a sticker, information about the
sticker
    :type sticker: :class:`telebot.types.Sticker`

    :param video: Optional. Message is a video, information about the video
    :type video: :class:`telebot.types.Video`

    :param video_note: Optional. Message is a video note, information about
the video message
    :type video_note: :class:`telebot.types.VideoNote`

    :param voice: Optional. Message is a voice message, information about the
file
    :type voice: :class:`telebot.types.Voice`

    :param caption: Optional. Caption for the animation, audio, document,
photo, video or voice
    :type caption: :obj:`str`

    :param caption_entities: Optional. For messages with a caption, special
entities like usernames, URLs, bot
        commands, etc. that appear in the caption
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param contact: Optional. Message is a shared contact, information about
the contact
    :type contact: :class:`telebot.types.Contact`

    :param dice: Optional. Message is a dice with random value
    :type dice: :class:`telebot.types.Dice`

    :param game: Optional. Message is a game, information about the game.
More about games »
    :type game: :class:`telebot.types.Game`

    :param poll: Optional. Message is a native poll, information about the
poll
    :type poll: :class:`telebot.types.Poll`

    :param venue: Optional. Message is a venue, information about the venue.
For backward compatibility, when this
        field is set, the location field will also be set
    :type venue: :class:`telebot.types.Venue`

    :param location: Optional. Message is a shared location, information
about the location
    :type location: :class:`telebot.types.Location`

    :param new_chat_members: Optional. New members that were added to the
group or supergroup and information about
        them (the bot itself may be one of these members)
    :type new_chat_members: :obj:`list` of :class:`telebot.types.User`
```

```
    :param left_chat_member: Optional. A member was removed from the group,
information about them (this member may be
        the bot itself)
    :type left_chat_member: :class:`telebot.types.User`

    :param new_chat_title: Optional. A chat title was changed to this value
    :type new_chat_title: :obj:`str`

    :param new_chat_photo: Optional. A chat photo was change to this value
    :type new_chat_photo: :obj:`list` of :class:`telebot.types.PhotoSize`

    :param delete_chat_photo: Optional. Service message: the chat photo was
deleted
    :type delete_chat_photo: :obj:`bool`

    :param group_chat_created: Optional. Service message: the group has been
created
    :type group_chat_created: :obj:`bool`

    :param supergroup_chat_created: Optional. Service message: the supergroup
has been created. This field can't be
        received in a message coming through updates, because bot can't be a
member of a supergroup when it is created. It can
        only be found in reply_to_message if someone replies to a very first
message in a directly created supergroup.
    :type supergroup_chat_created: :obj:`bool`

    :param channel_chat_created: Optional. Service message: the channel has
been created. This field can't be
        received in a message coming through updates, because bot can't be a
member of a channel when it is created. It can only
        be found in reply_to_message if someone replies to a very first
message in a channel.
    :type channel_chat_created: :obj:`bool`

    :param message_auto_delete_timer_changed: Optional. Service message:
auto-delete timer settings changed in
        the chat
    :type message_auto_delete_timer_changed:
:class:`telebot.types.MessageAutoDeleteTimerChanged`

    :param migrate_to_chat_id: Optional. The group has been migrated to a
supergroup with the specified identifier.
        This number may have more than 32 significant bits and some
programming languages may have difficulty/silent
        defects in interpreting it. But it has at most 52 significant bits,
so a signed 64-bit integer or double-precision
        float type are safe for storing this identifier.
    :type migrate_to_chat_id: :obj:`int`

    :param migrate_from_chat_id: Optional. The supergroup has been migrated
from a group with the specified
        identifier. This number may have more than 32 significant bits and
some programming languages may have
        difficulty/silent defects in interpreting it. But it has at most 52
significant bits, so a signed 64-bit integer or
        double-precision float type are safe for storing this identifier.
    :type migrate_from_chat_id: :obj:`int`

    :param pinned_message: Optional. Specified message was pinned. Note that
the Message object in this field will not
        contain further reply_to_message fields even if it is itself a reply.
    :type pinned_message: :class:`telebot.types.Message`
```

```
    :param invoice: Optional. Message is an invoice for a payment,
information about the invoice. More about payments »
    :type invoice: :class:`telebot.types.Invoice`

    :param successful_payment: Optional. Message is a service message about a
successful payment, information about
        the payment. More about payments »
    :type successful_payment: :class:`telebot.types.SuccessfulPayment`

    :param connected_website: Optional. The domain name of the website on
which the user has logged in. More about
        Telegram Login »
    :type connected_website: :obj:`str`

    :param passport_data: Optional. Telegram Passport data
    :type passport_data: :class:`telebot.types.PassportData`

    :param proximity_alert_triggered: Optional. Service message. A user in
the chat triggered another user's
        proximity alert while sharing Live Location.
    :type proximity_alert_triggered:
:class:`telebot.types.ProximityAlertTriggered`

    :param forum_topic_created: Optional. Service message: forum topic
created
    :type forum_topic_created: :class:`telebot.types.ForumTopicCreated`

    :param forum_topic_closed: Optional. Service message: forum topic closed
    :type forum_topic_closed: :class:`telebot.types.ForumTopicClosed`

    :param forum_topic_reopened: Optional. Service message: forum topic
reopened
    :type forum_topic_reopened: :class:`telebot.types.ForumTopicReopened`

    :param video_chat_scheduled: Optional. Service message: video chat
scheduled
    :type video_chat_scheduled: :class:`telebot.types.VideoChatScheduled`

    :param video_chat_started: Optional. Service message: video chat started
    :type video_chat_started: :class:`telebot.types.VideoChatStarted`

    :param video_chat_ended: Optional. Service message: video chat ended
    :type video_chat_ended: :class:`telebot.types.VideoChatEnded`

    :param video_chat_participants_invited: Optional. Service message: new
participants invited to a video chat
    :type video_chat_participants_invited:
:class:`telebot.types.VideoChatParticipantsInvited`

    :param web_app_data: Optional. Service message: data sent by a Web App
    :type web_app_data: :class:`telebot.types.WebAppData`

    :param reply_markup: Optional. Inline keyboard attached to the message.
login_url buttons are represented as
        ordinary url buttons.
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Message`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
```

```
        obj = cls.check_json(json_string, dict_copy=False)
        message_id = obj['message_id']
        from_user = User.de_json(obj.get('from'))
        date = obj['date']
        chat = Chat.de_json(obj['chat'])
        content_type = None
        opts = {}
        if 'sender_chat' in obj:
            opts['sender_chat'] = Chat.de_json(obj['sender_chat'])
        if 'forward_from' in obj:
            opts['forward_from'] = User.de_json(obj['forward_from'])
        if 'forward_from_chat' in obj:
            opts['forward_from_chat'] =
Chat.de_json(obj['forward_from_chat'])
        if 'forward_from_message_id' in obj:
            opts['forward_from_message_id'] =
obj.get('forward_from_message_id')
        if 'forward_signature' in obj:
            opts['forward_signature'] = obj.get('forward_signature')
        if 'forward_sender_name' in obj:
            opts['forward_sender_name'] = obj.get('forward_sender_name')
        if 'forward_date' in obj:
            opts['forward_date'] = obj.get('forward_date')
        if 'is_automatic_forward' in obj:
            opts['is_automatic_forward'] = obj.get('is_automatic_forward')
        if 'is_topic_message' in obj:
            opts['is_topic_message'] = obj.get('is_topic_message')
        if 'message_thread_id' in obj:
            opts['message_thread_id'] = obj.get('message_thread_id')
        if 'reply_to_message' in obj:
            opts['reply_to_message'] =
Message.de_json(obj['reply_to_message'])
        if 'via_bot' in obj:
            opts['via_bot'] = User.de_json(obj['via_bot'])
        if 'edit_date' in obj:
            opts['edit_date'] = obj.get('edit_date')
        if 'has_protected_content' in obj:
            opts['has_protected_content'] = obj.get('has_protected_content')
        if 'media_group_id' in obj:
            opts['media_group_id'] = obj.get('media_group_id')
        if 'author_signature' in obj:
            opts['author_signature'] = obj.get('author_signature')
        if 'text' in obj:
            opts['text'] = obj['text']
            content_type = 'text'
        if 'entities' in obj:
            opts['entities'] = Message.parse_entities(obj['entities'])
        if 'caption_entities' in obj:
            opts['caption_entities'] =
Message.parse_entities(obj['caption_entities'])
        if 'audio' in obj:
            opts['audio'] = Audio.de_json(obj['audio'])
            content_type = 'audio'
        if 'document' in obj:
            opts['document'] = Document.de_json(obj['document'])
            content type = 'document'
        if 'animation' in obj:
            # Document content type accompanies "animation",
            # so "animation" should be checked below "document" to override
it
            opts['animation'] = Animation.de_json(obj['animation'])
            content_type = 'animation'
        if 'game' in obj:
            opts['game'] = Game.de_json(obj['game'])
```

```
            content_type = 'game'
        if 'photo' in obj:
            opts['photo'] = Message.parse_photo(obj['photo'])
            content_type = 'photo'
        if 'sticker' in obj:
            opts['sticker'] = Sticker.de_json(obj['sticker'])
            content_type = 'sticker'
        if 'video' in obj:
            opts['video'] = Video.de_json(obj['video'])
            content_type = 'video'
        if 'video_note' in obj:
            opts['video_note'] = VideoNote.de_json(obj['video_note'])
            content_type = 'video_note'
        if 'voice' in obj:
            opts['voice'] = Audio.de_json(obj['voice'])
            content_type = 'voice'
        if 'caption' in obj:
            opts['caption'] = obj['caption']
        if 'contact' in obj:
            opts['contact'] = Contact.de_json(json.dumps(obj['contact']))
            content_type = 'contact'
        if 'location' in obj:
            opts['location'] = Location.de_json(obj['location'])
            content_type = 'location'
        if 'venue' in obj:
            opts['venue'] = Venue.de_json(obj['venue'])
            content_type = 'venue'
        if 'dice' in obj:
            opts['dice'] = Dice.de_json(obj['dice'])
            content_type = 'dice'
        if 'new_chat_members' in obj:
            new_chat_members = []
            for member in obj['new_chat_members']:
                new_chat_members.append(User.de_json(member))
            opts['new_chat_members'] = new_chat_members
            content_type = 'new_chat_members'
        if 'left_chat_member' in obj:
            opts['left_chat_member'] = User.de_json(obj['left_chat_member'])
            content_type = 'left_chat_member'
        if 'new_chat_title' in obj:
            opts['new_chat_title'] = obj['new_chat_title']
            content_type = 'new_chat_title'
        if 'new_chat_photo' in obj:
            opts['new_chat_photo'] =
Message.parse_photo(obj['new_chat_photo'])
            content_type = 'new_chat_photo'
        if 'delete_chat_photo' in obj:
            opts['delete_chat_photo'] = obj['delete_chat_photo']
            content_type = 'delete_chat_photo'
        if 'group_chat_created' in obj:
            opts['group_chat_created'] = obj['group_chat_created']
            content_type = 'group_chat_created'
        if 'supergroup_chat_created' in obj:
            opts['supergroup_chat_created'] = obj['supergroup_chat_created']
            content_type = 'supergroup_chat_created'
        if 'channel_chat_created' in obj:
            opts['channel_chat_created'] = obj['channel_chat_created']
            content_type = 'channel_chat_created'
        if 'migrate_to_chat_id' in obj:
            opts['migrate_to_chat_id'] = obj['migrate_to_chat_id']
            content_type = 'migrate_to_chat_id'
        if 'migrate_from_chat_id' in obj:
            opts['migrate_from_chat_id'] = obj['migrate_from_chat_id']
            content_type = 'migrate_from_chat_id'
```

```
        if 'pinned_message' in obj:
            opts['pinned_message'] = Message.de_json(obj['pinned_message'])
            content_type = 'pinned_message'
        if 'invoice' in obj:
            opts['invoice'] = Invoice.de_json(obj['invoice'])
            content_type = 'invoice'
        if 'successful_payment' in obj:
            opts['successful_payment'] =
SuccessfulPayment.de_json(obj['successful_payment'])
            content_type = 'successful_payment'
        if 'connected_website' in obj:
            opts['connected_website'] = obj['connected_website']
            content_type = 'connected_website'
        if 'poll' in obj:
            opts['poll'] = Poll.de_json(obj['poll'])
            content_type = 'poll'
        if 'passport_data' in obj:
            opts['passport_data'] = obj['passport_data']
            content_type = 'passport_data'
        if 'proximity_alert_triggered' in obj:
            opts['proximity_alert_triggered'] =
ProximityAlertTriggered.de_json(obj[
                'proximity_alert_triggered'])
            content_type = 'proximity_alert_triggered'
        if 'video_chat_scheduled' in obj:
            opts['video_chat_scheduled'] =
VideoChatScheduled.de_json(obj['video_chat_scheduled'])
            opts['voice_chat_scheduled'] = opts['video_chat_scheduled']  #
deprecated, for backward compatibility
            content_type = 'video_chat_scheduled'
        if 'video_chat_started' in obj:
            opts['video_chat_started'] =
VideoChatStarted.de_json(obj['video_chat_started'])
            opts['voice_chat_started'] = opts['video_chat_started']  #
deprecated, for backward compatibility
            content_type = 'video_chat_started'
        if 'video_chat_ended' in obj:
            opts['video_chat_ended'] =
VideoChatEnded.de_json(obj['video_chat_ended'])
            opts['voice_chat_ended'] = opts['video_chat_ended']  #
deprecated, for backward compatibility
            content_type = 'video_chat_ended'
        if 'video_chat_participants_invited' in obj:
            opts['video_chat_participants_invited'] =
VideoChatParticipantsInvited.de_json(obj['video_chat_participants_invited'])
            opts['voice_chat_participants_invited'] =
opts['video_chat_participants_invited']  # deprecated, for backward
compatibility
            content_type = 'video_chat_participants_invited'
        if 'web_app_data' in obj:
            opts['web_app_data'] = WebAppData.de_json(obj['web_app_data'])
            content_type = 'web_app_data'
        if 'message_auto_delete_timer_changed' in obj:
            opts['message_auto_delete_timer_changed'] =
MessageAutoDeleteTimerChanged.de_json(obj['message_auto_delete_timer_changed'
])
            content_type = 'message_auto_delete_timer_changed'
        if 'reply_markup' in obj:
            opts['reply_markup'] =
InlineKeyboardMarkup.de_json(obj['reply_markup'])
        if 'forum_topic_created' in obj:
            opts['forum_topic_created'] =
ForumTopicCreated.de_json(obj['forum_topic_created'])
            content_type = 'forum_topic_created'
```

```
        if 'forum_topic_closed' in obj:
            opts['forum_topic_closed'] =
ForumTopicClosed.de_json(obj['forum_topic_closed'])
            content_type = 'forum_topic_closed'
        if 'forum_topic_reopened' in obj:
            opts['forum_topic_reopened'] =
ForumTopicReopened.de_json(obj['forum_topic_reopened'])
            content_type = 'forum_topic_reopened'
        return cls(message_id, from_user, date, chat, content_type, opts,
json_string)

    @classmethod
    def parse_chat(cls, chat):
        """
        Parses chat.
        """
        if 'first_name' not in chat:
            return GroupChat.de_json(chat)
        else:
            return User.de_json(chat)

    @classmethod
    def parse_photo(cls, photo_size_array):
        """
        Parses photo array.
        """
        ret = []
        for ps in photo_size_array:
            ret.append(PhotoSize.de_json(ps))
        return ret

    @classmethod
    def parse_entities(cls, message_entity_array):
        """
        Parses message entity array.
        """
        ret = []
        for me in message_entity_array:
            ret.append(MessageEntity.de_json(me))
        return ret

    def __init__(self, message_id, from_user, date, chat, content_type,
options, json_string):
        self.content_type: str = content_type
        self.id: int = message_id           # Lets fix the telegram usability
####up with ID in Message :)
        self.message_id: int = message_id
        self.from_user: User = from_user
        self.date: int = date
        self.chat: Chat = chat
        self.sender_chat: Optional[Chat] = None
        self.forward_from: Optional[User] = None
        self.forward_from_chat: Optional[Chat] = None
        self.forward_from_message_id: Optional[int] = None
        self.forward_signature: Optional[str] = None
        self.forward_sender_name: Optional[str] = None
        self.forward_date: Optional[int] = None
        self.is_automatic_forward: Optional[bool] = None
        self.reply_to_message: Optional[Message] = None
        self.via_bot: Optional[User] = None
        self.edit_date: Optional[int] = None
        self.has_protected_content: Optional[bool] = None
        self.media_group_id: Optional[str] = None
        self.author_signature: Optional[str] = None
```

```
        self.text: Optional[str] = None
        self.entities: Optional[List[MessageEntity]] = None
        self.caption_entities: Optional[List[MessageEntity]] = None
        self.audio: Optional[Audio] = None
        self.document: Optional[Document] = None
        self.photo: Optional[List[PhotoSize]] = None
        self.sticker: Optional[Sticker] = None
        self.video: Optional[Video] = None
        self.video_note: Optional[VideoNote] = None
        self.voice: Optional[Voice] = None
        self.caption: Optional[str] = None
        self.contact: Optional[Contact] = None
        self.location: Optional[Location] = None
        self.venue: Optional[Venue] = None
        self.animation: Optional[Animation] = None
        self.dice: Optional[Dice] = None
        self.new_chat_member: Optional[User] = None  # Deprecated since Bot
API 3.0. Not processed anymore
        self.new_chat_members: Optional[List[User]] = None
        self.left_chat_member: Optional[User] = None
        self.new_chat_title: Optional[str] = None
        self.new_chat_photo: Optional[List[PhotoSize]] = None
        self.delete_chat_photo: Optional[bool] = None
        self.group_chat_created: Optional[bool] = None
        self.supergroup_chat_created: Optional[bool] = None
        self.channel_chat_created: Optional[bool] = None
        self.migrate_to_chat_id: Optional[int] = None
        self.migrate_from_chat_id: Optional[int] = None
        self.pinned_message: Optional[Message] = None
        self.invoice: Optional[Invoice] = None
        self.successful_payment: Optional[SuccessfulPayment] = None
        self.connected_website: Optional[str] = None
        self.reply_markup: Optional[InlineKeyboardMarkup] = None
        self.message_thread_id: Optional[int] = None
        self.is_topic_message: Optional[bool] = None
        self.forum_topic_created: Optional[ForumTopicCreated] = None
        self.forum_topic_closed: Optional[ForumTopicClosed] = None
        self.forum_topic_reopened: Optional[ForumTopicReopened] = None
        for key in options:
            setattr(self, key, options[key])
        self.json = json_string

    def __html_text(self, text, entities):
        """
        Author: @sviat9440
        Updaters: @badiboy
        Message: "*Test* parse  formatting , [url](https://example.com),
[text_mention](tg://user?id=123456) and mention @username"

        .. code-block:: python3
            :caption: Example:

            message.html_text
            >> "<b>Test</b> parse <i>formatting</i>, <a
href=\"https://example.com\">url</a>, <a
href=\"tg://user?id=123456\">text mention</a> and mention @username"

        Custom subs:
            You can customize the substitutes. By default, there is no
substitute for the entities: hashtag, bot command, email. You can add or
modify substitute an existing entity.
        .. code-block:: python3
            :caption: Example:
```

```
            message.custom_subs = {"bold": "<strong
class=\"example\">{text}</strong>", "italic": "<i
class=\"example\">{text}</i>", "mention": "<a href={url}>{text}</a>"}
            message.html_text
            >> "<strong class=\"example\">Test</strong> parse <i
class=\"example\">formatting</i>, <a href=\"https://example.com\">url</a> and
<a href=\"tg://user?id=123456\">text mention</a> and mention <a
href=\"https://t.me/username\">@username</a>"
        """

        if not entities:
            return text

        _subs = {
            "bold": "<b>{text}</b>",
            "italic": "<i>{text}</i>",
            "pre": "<pre>{text}</pre>",
            "code": "<code>{text}</code>",
            # "url": "<a href=\"{url}\">{text}</a>", # @badiboy plain URLs
have no text and do not need tags
            "text_link": "<a href=\"{url}\">{text}</a>",
            "strikethrough": "<s>{text}</s>",
            "underline":     "<u>{text}</u>",
            "spoiler": "<span class=\"tg-spoiler\">{text}</span>",
        }

        if hasattr(self, "custom_subs"):
            for key, value in self.custom_subs.items():
                _subs[key] = value
        utf16_text = text.encode("utf-16-le")
        html_text = ""

        def func(upd_text, subst_type=None, url=None, user=None):
            upd_text = upd_text.decode("utf-16-le")
            if subst_type == "text_mention":
                subst_type = "text_link"
                url = "tg://user?id={0}".format(user.id)
            elif subst_type == "mention":
                url = "https://t.me/{0}".format(upd_text[1:])
            upd_text = upd_text.replace("&", "&amp;").replace("<",
"&lt;").replace(">", "&gt;")
            if not subst_type or not _subs.get(subst_type):
                return upd_text
            subs = _subs.get(subst_type)
            return subs.format(text=upd_text, url=url)

        offset = 0
        for entity in entities:
            if entity.offset > offset:
                html_text += func(utf16_text[offset * 2 : entity.offset * 2])
                offset = entity.offset
                html_text += func(utf16_text[offset * 2 : (offset +
entity.length) * 2], entity.type, entity.url, entity.user)
                offset += entity.length
            elif entity.offset == offset:
                html_text += func(utf16_text[offset * 2 : (offset +
entity.length) * 2], entity.type, entity.url, entity.user)
                offset += entity.length
            else:
                # Here we are processing nested entities.
                # We shouldn't update offset, because they are the same as
entity before.
                # And, here we are replacing previous string with a new html-
rendered text(previous string is already html-rendered,
```

```
                # And we don't change it).
                entity_string = utf16_text[entity.offset * 2 : (entity.offset
+ entity.length) * 2]
                formatted_string = func(entity_string, entity.type,
entity.url, entity.user)
                entity_string_decoded = entity_string.decode("utf-16-le")
                last_occurence = html_text.rfind(entity_string_decoded)
                string_length = len(entity_string_decoded)
                #html_text =
html_text.replace(html_text[last_occurence:last_occurence+string_length],
formatted_string)
                html_text = html_text[:last_occurence] + formatted_string +
html_text[last_occurence+string_length:]
        if offset * 2 < len(utf16_text):
            html_text += func(utf16_text[offset * 2:])

        return html_text

    @property
    def html_text(self):
        """
        Returns html-rendered text.
        """
        return self.__html_text(self.text, self.entities)

    @property
    def html_caption(self):
        """
        Returns html-rendered caption.
        """
        return self.__html_text(self.caption, self.caption_entities)


class MessageEntity(Dictionaryable, JsonSerializable, JsonDeserializable):
    """
    This object represents one special entity in a text message. For example,
hashtags, usernames, URLs, etc.

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

    :param type: Type of the entity. Currently, can be “mention” (@username),
“hashtag” (#hashtag), “cashtag”
        ($USD), “bot_command” (/start@jobs_bot), “url”
(https://telegram.org), “email”
        (do-not-reply@telegram.org), “phone_number” (+1-212-555-0123), “bold”
(bold text), “italic” (italic text),
        “underline” (underlined text), “strikethrough” (strikethrough text),
“spoiler” (spoiler message), “code”
        (monowidth string), “pre” (monowidth block), “text_link” (for
clickable text URLs), “text_mention” (for users
        without usernames), “custom emoji” (for inline custom emoji stickers)
    :type type: :obj:`str`

    :param offset: Offset in UTF-16 code units to the start of the entity
    :type offset: :obj:`int`

    :param length: Length of the entity in UTF-16 code units
    :type length: :obj:`int`

    :param url: Optional. For “text_link” only, URL that will be opened after
user taps on the text
    :type url: :obj:`str`

    :param user: Optional. For “text_mention” only, the mentioned user
```

```
    :type user: :class:`telebot.types.User`

    :param language: Optional. For “pre” only, the programming language of
the entity text
    :type language: :obj:`str`

    :param custom_emoji_id: Optional. For “custom_emoji” only, unique
identifier of the custom emoji.
        Use get_custom_emoji_stickers to get full information about the
sticker.
    :type custom_emoji_id: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.MessageEntity`
    """
    @staticmethod
    def to_list_of_dicts(entity_list) -> Union[List[Dict], None]:
        """
        Converts a list of MessageEntity objects to a list of dictionaries.
        """
        res = []
        for e in entity_list:
            res.append(MessageEntity.to_dict(e))
        return res or None

    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'user' in obj:
            obj['user'] = User.de_json(obj['user'])
        return cls(**obj)

    def __init__(self, type, offset, length, url=None, user=None,
language=None, custom_emoji_id=None, **kwargs):
        self.type: str = type
        self.offset: int = offset
        self.length: int = length
        self.url: str = url
        self.user: User = user
        self.language: str = language
        self.custom_emoji_id: str = custom_emoji_id

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {"type": self.type,
                "offset": self.offset,
                "length": self.length,
                "url": self.url,
                "user": self.user,
                "language":  self.language,
                "custom_emoji_id": self.custom_emoji_id}


class Dice(JsonSerializable, Dictionaryable, JsonDeserializable):
    """
    This object represents an animated emoji that displays a random value.

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

    :param emoji: Emoji on which the dice throw animation is based
    :type emoji: :obj:`str`
```

```
    :param value: Value of the dice, 1-6 for "🎲", "🎯" and "🎳" base emoji,
1-5 for "🏀" and "⚽" base emoji, 1-64 for "🎰" base emoji
    :type value: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Dice`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, value, emoji, **kwargs):
        self.value: int = value
        self.emoji: str = emoji

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {'value': self.value,
                'emoji': self.emoji}


class PhotoSize(JsonDeserializable):
    """
    This object represents one size of a photo or a file / sticker thumbnail.

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

    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`

    :param width: Photo width
    :type width: :obj:`int`

    :param height: Photo height
    :type height: :obj:`int`

    :param file_size: Optional. File size in bytes
    :type file_size: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.PhotoSize`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, width, height,
file_size=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
```

```
        self.width: int = width
        self.height: int = height
        self.file_size: int = file_size


class Audio(JsonDeserializable):
    """
    This object represents an audio file to be treated as music by the
Telegram clients.

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

    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`

    :param duration: Duration of the audio in seconds as defined by sender
    :type duration: :obj:`int`

    :param performer: Optional. Performer of the audio as defined by sender
or by audio tags
    :type performer: :obj:`str`

    :param title: Optional. Title of the audio as defined by sender or by
audio tags
    :type title: :obj:`str`

    :param file_name: Optional. Original filename as defined by sender
    :type file_name: :obj:`str`

    :param mime_type: Optional. MIME type of the file as defined by sender
    :type mime_type: :obj:`str`

    :param file_size: Optional. File size in bytes. It can be bigger than
2^31 and some programming languages may have
        difficulty/silent defects in interpreting it. But it has at most 52
significant bits, so a signed 64-bit integer or
        double-precision float type are safe for storing this value.
    :type file_size: :obj:`int`

    :param thumb: Optional. Thumbnail of the album cover to which the music
file belongs
    :type thumb: :class:`telebot.types.PhotoSize`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Audio`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'thumb' in obj and 'file_id' in obj['thumb']:
            obj['thumb'] = PhotoSize.de_json(obj['thumb'])
        else:
            obj['thumb'] = None
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, duration, performer=None,
title=None, file_name=None, mime_type=None,
```

```
                 file_size=None, thumb=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.duration: int = duration
        self.performer: str = performer
        self.title: str = title
        self.file_name: str = file_name
        self.mime_type: str = mime_type
        self.file_size: int = file_size
        self.thumb: PhotoSize = thumb


class Voice(JsonDeserializable):
    """
    This object represents a voice note.

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

    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`

    :param duration: Duration of the audio in seconds as defined by sender
    :type duration: :obj:`int`

    :param mime_type: Optional. MIME type of the file as defined by sender
    :type mime_type: :obj:`str`

    :param file_size: Optional. File size in bytes. It can be bigger than
2^31 and some programming languages may have
        difficulty/silent defects in interpreting it. But it has at most 52
significant bits, so a signed 64-bit integer or
        double-precision float type are safe for storing this value.
    :type file_size: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Voice`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, duration, mime_type=None,
file_size=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.duration: int = duration
        self.mime_type: str = mime_type
        self.file_size: int = file_size


class Document(JsonDeserializable):
    """
    This object represents a general file (as opposed to photos, voice
messages and audio files).

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

```
    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`

    :param thumb: Optional. Document thumbnail as defined by sender
    :type thumb: :class:`telebot.types.PhotoSize`

    :param file_name: Optional. Original filename as defined by sender
    :type file_name: :obj:`str`

    :param mime_type: Optional. MIME type of the file as defined by sender
    :type mime_type: :obj:`str`

    :param file_size: Optional. File size in bytes. It can be bigger than
2^31 and some programming languages may have
        difficulty/silent defects in interpreting it. But it has at most 52
significant bits, so a signed 64-bit integer or
        double-precision float type are safe for storing this value.
    :type file_size: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Document`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'thumb' in obj and 'file_id' in obj['thumb']:
            obj['thumb'] = PhotoSize.de_json(obj['thumb'])
        else:
            obj['thumb'] = None
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, thumb=None, file_name=None,
mime_type=None, file_size=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.thumb: PhotoSize = thumb
        self.file_name: str = file_name
        self.mime_type: str = mime_type
        self.file_size: int = file_size


class Video(JsonDeserializable):
    """
    This object represents a video file.

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

    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`
```

```
    :param width: Video width as defined by sender
    :type width: :obj:`int`

    :param height: Video height as defined by sender
    :type height: :obj:`int`

    :param duration: Duration of the video in seconds as defined by sender
    :type duration: :obj:`int`

    :param thumb: Optional. Video thumbnail
    :type thumb: :class:`telebot.types.PhotoSize`

    :param file_name: Optional. Original filename as defined by sender
    :type file_name: :obj:`str`

    :param mime_type: Optional. MIME type of the file as defined by sender
    :type mime_type: :obj:`str`

    :param file_size: Optional. File size in bytes. It can be bigger than 2^31 and some programming languages may have
        difficulty/silent defects in interpreting it. But it has at most 52 significant bits, so a signed 64-bit integer or
        double-precision float type are safe for storing this value.
    :type file_size: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Video`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'thumb' in obj and 'file_id' in obj['thumb']:
            obj['thumb'] = PhotoSize.de_json(obj['thumb'])
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, width, height, duration, thumb=None, file_name=None, mime_type=None, file_size=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.width: int = width
        self.height: int = height
        self.duration: int = duration
        self.thumb: PhotoSize = thumb
        self.file_name: str = file_name
        self.mime_type: str = mime_type
        self.file_size: int = file_size


class VideoNote(JsonDeserializable):
    """
    This object represents a video message (available in Telegram apps as of v.4.0).

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

    :param file_id: Identifier for this file, which can be used to download or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`
```

```
    :param length: Video width and height (diameter of the video message) as
defined by sender
    :type length: :obj:`int`

    :param duration: Duration of the video in seconds as defined by sender
    :type duration: :obj:`int`

    :param thumb: Optional. Video thumbnail
    :type thumb: :class:`telebot.types.PhotoSize`

    :param file_size: Optional. File size in bytes
    :type file_size: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.VideoNote`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'thumb' in obj and 'file_id' in obj['thumb']:
            obj['thumb'] = PhotoSize.de_json(obj['thumb'])
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, length, duration, thumb=None,
file_size=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.length: int = length
        self.duration: int = duration
        self.thumb: PhotoSize = thumb
        self.file_size: int = file_size


class Contact(JsonDeserializable):
    """
    This object represents a phone contact.

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

    :param phone_number: Contact's phone number
    :type phone_number: :obj:`str`

    :param first_name: Contact's first name
    :type first_name: :obj:`str`

    :param last_name: Optional. Contact's last name
    :type last_name: :obj:`str`

    :param user_id: Optional. Contact's user identifier in Telegram. This
number may have more than 32 significant bits
        and some programming languages may have difficulty/silent defects in
interpreting it. But it has at most 52
        significant bits, so a 64-bit integer or double-precision float type
are safe for storing this identifier.
    :type user_id: :obj:`int`

    :param vcard: Optional. Additional data about the contact in the form of
a vCard
    :type vcard: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Contact`
```

```
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, phone_number, first_name, last_name=None,
user_id=None, vcard=None, **kwargs):
        self.phone_number: str = phone_number
        self.first_name: str = first_name
        self.last_name: str = last_name
        self.user_id: int = user_id
        self.vcard: str = vcard


class Location(JsonDeserializable, JsonSerializable, Dictionaryable):
    """
    This object represents a point on the map.

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

    :param longitude: Longitude as defined by sender
    :type longitude: :obj:`float`

    :param latitude: Latitude as defined by sender
    :type latitude: :obj:`float`

    :param horizontal_accuracy: Optional. The radius of uncertainty for the
location, measured in meters; 0-1500
    :type horizontal_accuracy: :obj:`float` number

    :param live_period: Optional. Time relative to the message sending date,
during which the location can be updated;
        in seconds. For active live locations only.
    :type live_period: :obj:`int`

    :param heading: Optional. The direction in which user is moving, in
degrees; 1-360. For active live locations only.
    :type heading: :obj:`int`

    :param proximity_alert_radius: Optional. The maximum distance for
proximity alerts about approaching another
        chat member, in meters. For sent live locations only.
    :type proximity_alert_radius: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Location`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def   init  (self, longitude, latitude, horizontal accuracy=None,
                 live_period=None, heading=None, proximity_alert_radius=None,
**kwargs):
        self.longitude: float = longitude
        self.latitude: float = latitude
        self.horizontal_accuracy: float = horizontal_accuracy
        self.live_period: int = live_period
        self.heading: int = heading
        self.proximity_alert_radius: int = proximity_alert_radius
```

```
    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {
            "longitude": self.longitude,
            "latitude": self.latitude,
            "horizontal_accuracy": self.horizontal_accuracy,
            "live_period": self.live_period,
            "heading": self.heading,
            "proximity_alert_radius": self.proximity_alert_radius,
        }


class Venue(JsonDeserializable):
    """
    This object represents a venue.

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

    :param location: Venue location. Can't be a live location
    :type location: :class:`telebot.types.Location`

    :param title: Name of the venue
    :type title: :obj:`str`

    :param address: Address of the venue
    :type address: :obj:`str`

    :param foursquare_id: Optional. Foursquare identifier of the venue
    :type foursquare_id: :obj:`str`

    :param foursquare_type: Optional. Foursquare type of the venue. (For
example, “arts_entertainment/default”,
        “arts_entertainment/aquarium” or “food/icecream”.)
    :type foursquare_type: :obj:`str`

    :param google_place_id: Optional. Google Places identifier of the venue
    :type google_place_id: :obj:`str`

    :param google_place_type: Optional. Google Places type of the venue. (See
supported types.)
    :type google_place_type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Venue`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        obj['location'] = Location.de_json(obj['location'])
        return cls(**obj)

    def __init__(self, location, title, address, foursquare_id=None,
foursquare_type=None,
                 google_place_id=None, google_place_type=None, **kwargs):
        self.location: Location = location
        self.title: str = title
        self.address: str = address
        self.foursquare_id: str = foursquare_id
        self.foursquare_type: str = foursquare_type
        self.google_place_id: str = google_place_id
```

```
        self.google_place_type: str = google_place_type


class UserProfilePhotos(JsonDeserializable):
    """
    This object represent a user's profile pictures.

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

    :param total_count: Total number of profile pictures the target user has
    :type total_count: :obj:`int`

    :param photos: Requested profile pictures (in up to 4 sizes each)
    :type photos: :obj:`list` of :obj:`list` of
:class:`telebot.types.PhotoSize`

    :return: Instance of the class
    :rtype: :class:`telebot.types.UserProfilePhotos`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'photos' in obj:
            photos = [[PhotoSize.de_json(y) for y in x] for x in
obj['photos']]
            obj['photos'] = photos
        return cls(**obj)

    def __init__(self, total_count, photos=None, **kwargs):
        self.total_count: int = total_count
        self.photos: List[PhotoSize] = photos


class File(JsonDeserializable):
    """
    This object represents a file ready to be downloaded. The file can be
downloaded via the link https://api.telegram.org/file/bot<token>/<file_path>.
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 getFile.

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

    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`

    :param file_size: Optional. File size in bytes. It can be bigger than
2^31 and some programming languages may have
        difficulty/silent defects in interpreting it. But it has at most 52
significant bits, so a signed 64-bit integer or
        double-precision float type are safe for storing this value.
    :type file_size: :obj:`int`

    :param file_path: Optional. File path. Use
https://api.telegram.org/file/bot<token>/<file_path> to get the
        file.
    :type file_path: :obj:`str`
```

```
    :return: Instance of the class
    :rtype: :class:`telebot.types.File`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, file_size=None,
file_path=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.file_size: int = file_size
        self.file_path: str = file_path


class ForceReply(JsonSerializable):
    """
    Upon receiving a message with this object, Telegram clients will display
a reply interface to the user (act as if the user has selected the bot's
message and tapped 'Reply'). This can be extremely useful if you want to
create user-friendly step-by-step interfaces without having to sacrifice
privacy mode.

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

    :param force_reply: Shows reply interface to the user, as if they
manually selected the bot's message and tapped
        'Reply'
    :type force_reply: :obj:`bool`

    :param input_field_placeholder: Optional. The placeholder to be shown in
the input field when the reply is active;
        1-64 characters
    :type input_field_placeholder: :obj:`str`

    :param selective: Optional. Use this parameter if you want to force reply
from specific users only. Targets: 1) users
        that are @mentioned in the text of the Message object; 2) if the
bot's message is a reply (has reply to message id),
        sender of the original message.
    :type selective: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ForceReply`
    """
    def __init__(self, selective: Optional[bool]=None,
input_field_placeholder: Optional[str]=None):
        self.selective: bool = selective
        self.input_field_placeholder: str = input_field_placeholder

    def to_json(self):
        json_dict = {'force_reply': True}
        if self.selective is not None:
            json_dict['selective'] = self.selective
        if self.input_field_placeholder:
            json_dict['input_field_placeholder'] =
self.input_field_placeholder
        return json.dumps(json_dict)


class ReplyKeyboardRemove(JsonSerializable):
```

```
    """
    Upon receiving a message with this object, Telegram clients will remove
the current custom keyboard and display the default letter-keyboard. By
default, custom keyboards are displayed until a new keyboard is sent by a
bot. An exception is made for one-time keyboards that are hidden immediately
after the user presses a button (see ReplyKeyboardMarkup).

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

    :param remove_keyboard: Requests clients to remove the custom keyboard
(user will not be able to summon this
        keyboard; if you want to hide the keyboard from sight but keep it
accessible, use one_time_keyboard in
        ReplyKeyboardMarkup)
        Note that this parameter is set to True by default by the library.
You cannot modify it.
    :type remove_keyboard: :obj:`bool`

    :param selective: Optional. Use this parameter if you want to remove the
keyboard for specific users only. Targets:
        1) users that are @mentioned in the text of the Message object; 2) if
the bot's message is a reply (has
        reply to message id), sender of the original message.Example: A user
votes in a poll, bot returns confirmation
        message in reply to the vote and removes the keyboard for that user,
while still showing the keyboard with poll options
        to users who haven't voted yet.
    :type selective: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ReplyKeyboardRemove`
    """
    def __init__(self, selective=None):
        self.selective: bool = selective

    def to_json(self):
        json_dict = {'remove_keyboard': True}
        if self.selective:
            json_dict['selective'] = self.selective
        return json.dumps(json_dict)


class WebAppInfo(JsonDeserializable, Dictionaryable):
    """
    Describes a Web App.

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

    :param url: An HTTPS URL of a Web App to be opened with additional data
as specified in Initializing Web Apps
    :type url: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.WebAppInfo`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        return cls(**obj)

    def __init__(self, url, **kwargs):
        self.url: str = url
```

```
    def to_dict(self):
        return {'url': self.url}


class ReplyKeyboardMarkup(JsonSerializable):
    """
    This object represents a custom keyboard with reply options (see
Introduction to bots for details and examples).

    .. code-block:: python3
        :caption: Example on creating ReplyKeyboardMarkup object

        from telebot.types import ReplyKeyboardMarkup, KeyboardButton

        markup = ReplyKeyboardMarkup(resize keyboard=True)
        markup.add(KeyboardButton('Text'))
        # or:
        markup.add('Text')

        # display this markup:
        bot.send_message(chat_id, 'Text', reply_markup=markup)

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

    :param keyboard: :obj:`list` of button rows, each represented by an
:obj:`list` of
        :class:`telebot.types.KeyboardButton` objects
    :type keyboard: :obj:`list` of :obj:`list` of
:class:`telebot.types.KeyboardButton`

    :param resize_keyboard: Optional. Requests clients to resize the keyboard
vertically for optimal fit (e.g., make
        the keyboard smaller if there are just two rows of buttons). Defaults
to false, in which case the custom keyboard is
        always of the same height as the app's standard keyboard.
    :type resize_keyboard: :obj:`bool`

    :param one_time_keyboard: Optional. Requests clients to hide the keyboard
as soon as it's been used. The keyboard
        will still be available, but clients will automatically display the
usual letter-keyboard in the chat - the user can
        press a special button in the input field to see the custom keyboard
again. Defaults to false.
    :type one_time_keyboard: :obj:`bool`

    :param input_field_placeholder: Optional. The placeholder to be shown in
the input field when the keyboard is
        active; 1-64 characters
    :type input_field_placeholder: :obj:`str`

    :param selective: Optional. Use this parameter if you want to show the
keyboard to specific users only. Targets: 1)
        users that are @mentioned in the text of the Message object; 2) if
the bot's message is a reply (has
        reply_to_message_id), sender of the original message.Example: A user
requests to change the bot's language, bot
        replies to the request with a keyboard to select the new language.
Other users in the group don't see the keyboard.
    :type selective: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ReplyKeyboardMarkup`
```

```
    """
    max_row_keys = 12

    def __init__(self, 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):
        if row_width > self.max_row_keys:
            # Todo: Will be replaced with Exception in future releases
            if not DISABLE_KEYLEN_ERROR:
                logger.error('Telegram does not support reply keyboard row
width over %d.' % self.max_row_keys)
            row_width = self.max_row_keys

        self.resize_keyboard: bool = resize_keyboard
        self.one_time_keyboard: bool = one_time_keyboard
        self.selective: bool = selective
        self.row_width: int = row_width
        self.input_field_placeholder: str = input_field_placeholder
        self.keyboard: List[List[KeyboardButton]] = []

    def add(self, *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

        :param args: KeyboardButton to append to the keyboard
        :type args: :obj:`str` or :class:`telebot.types.KeyboardButton`

        :param row_width: width of row
        :type row_width: :obj:`int`

        :return: self, to allow function chaining.
        :rtype: :class:`telebot.types.ReplyKeyboardMarkup`
        """
        if row_width is None:
            row_width = self.row_width

        if row_width > self.max_row_keys:
            # Todo: Will be replaced with Exception in future releases
            if not DISABLE_KEYLEN_ERROR:
                logger.error('Telegram does not support reply keyboard row
width over %d.' % self.max_row_keys)
            row_width = self.max_row_keys

        for row in util.chunks(args, row_width):
            button_array = []
            for button in row:
                if util.is_string(button):
                    button_array.append({'text': button})
                elif util.is_bytes(button):
                    button_array.append({'text': button.decode('utf-8')})
                else:
                    button_array.append(button.to_dict())
            self.keyboard.append(button_array)

        return self
```

```
    def row(self, *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

        :param args: strings
        :type args: :obj:`str`

        :return: self, to allow function chaining.
        :rtype: :class:`telebot.types.ReplyKeyboardMarkup`
        """

        return self.add(*args, row_width=self.max_row_keys)

    def to_json(self):
        json_dict = {'keyboard': self.keyboard}
        if self.one_time_keyboard is not None:
            json_dict['one_time_keyboard'] = self.one_time_keyboard
        if self.resize_keyboard is not None:
            json_dict['resize_keyboard'] = self.resize_keyboard
        if self.selective is not None:
            json_dict['selective'] = self.selective
        if self.input_field_placeholder:
            json_dict['input_field_placeholder'] =
self.input_field_placeholder
        return json.dumps(json_dict)


class KeyboardButtonPollType(Dictionaryable):
    """
    This object represents type of a poll, which is allowed to be created and
sent when the corresponding button is pressed.

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

    :param type: Optional. If quiz is passed, the user will be allowed to
create only polls in the quiz mode. If regular is
        passed, only regular polls will be allowed. Otherwise, the user will
be allowed to create a poll of any type.
    :type type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.KeyboardButtonPollType`
    """
    def __init__(self, type=''):
        self.type: str = type

    def to_dict(self):
        return {'type': self.type}


class KeyboardButton(Dictionaryable, JsonSerializable):
    """
    This object represents one button of the reply keyboard. For simple text
buttons String can be used instead of this object to specify text of the
button. Optional fields web app, request contact, request location, and
request_poll are mutually exclusive.

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

```
    :param text: Text of the button. If none of the optional fields are used,
it will be sent as a message when the button is
        pressed
    :type text: :obj:`str`

    :param request_contact: Optional. If True, the user's phone number will
be sent as a contact when the button is
        pressed. Available in private chats only.
    :type request_contact: :obj:`bool`

    :param request_location: Optional. If True, the user's current location
will be sent when the button is pressed.
        Available in private chats only.
    :type request_location: :obj:`bool`

    :param request_poll: Optional. If specified, the user will be asked to
create a poll and send it to the bot when the
        button is pressed. Available in private chats only.
    :type request_poll: :class:`telebot.types.KeyboardButtonPollType`

    :param web_app: Optional. If specified, the described Web App will be
launched when the button is pressed. The Web App
        will be able to send a “web_app_data” service message. Available in
private chats only.
    :type web_app: :class:`telebot.types.WebAppInfo`

    :return: Instance of the class
    :rtype: :class:`telebot.types.KeyboardButton`
    """
    def __init__(self, text: str, request_contact: Optional[bool]=None,
            request_location: Optional[bool]=None, request_poll:
Optional[KeyboardButtonPollType]=None,
            web_app: WebAppInfo=None):
        self.text: str = text
        self.request_contact: bool = request_contact
        self.request_location: bool = request_location
        self.request_poll: KeyboardButtonPollType = request_poll
        self.web_app: WebAppInfo = web_app

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = {'text': self.text}
        if self.request_contact is not None:
            json_dict['request_contact'] = self.request_contact
        if self.request_location is not None:
            json_dict['request_location'] = self.request_location
        if self.request_poll is not None:
            json_dict['request_poll'] = self.request_poll.to_dict()
        if self.web_app is not None:
            json_dict['web_app'] = self.web_app.to_dict()
        return json_dict


class InlineKeyboardMarkup(Dictionaryable, JsonSerializable,
JsonDeserializable):
    """
    This object represents an inline keyboard that appears right next to the
message it belongs to.

    .. note::
        It is recommended to use :meth:`telebot.util.quick_markup` instead.
```

```
    .. code-block:: python3
        :caption: Example of a custom keyboard with buttons.

        from telebot.util import quick_markup

        markup = quick_markup(
            {'text': 'Press me', 'callback data': 'press'},
            {'text': 'Press me too', 'callback_data': 'press_too'}
        )

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

    :param inline_keyboard: :obj:`list` of button rows, each represented by
an :obj:`list` of
        :class:`telebot.types.InlineKeyboardButton` objects
    :type inline_keyboard: :obj:`list` of :obj:`list` of
:class:`telebot.types.InlineKeyboardButton`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineKeyboardMarkup`
    """
    max_row_keys = 8

    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        keyboard = [[InlineKeyboardButton.de_json(button) for button in row]
for row in obj['inline_keyboard']]
        return cls(keyboard = keyboard)

    def __init__(self, keyboard=None, row_width=3):
        if row_width > self.max_row_keys:
            # Todo: Will be replaced with Exception in future releases
            logger.error('Telegram does not support inline keyboard row width
over %d.' % self.max_row_keys)
            row_width = self.max_row_keys

        self.row_width: int = row_width
        self.keyboard: List[List[InlineKeyboardButton]] = keyboard or []

    def add(self, *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

        :param args: Array of InlineKeyboardButton to append to the keyboard
        :type args: :obj:`list` of
:class:`telebot.types.InlineKeyboardButton`

        :param row_width: width of row
        :type row_width: :obj:`int`

        :return: self, to allow function chaining.
        :rtype: :class:`telebot.types.InlineKeyboardMarkup`
        """
        if row_width is None:
```

```
            row_width = self.row_width

        if row_width > self.max_row_keys:
            # Todo: Will be replaced with Exception in future releases
            logger.error('Telegram does not support inline keyboard row width
over %d.' % self.max_row_keys)
            row_width = self.max_row_keys

        for row in util.chunks(args, row_width):
            button_array = [button for button in row]
            self.keyboard.append(button_array)

        return self

    def row(self, *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

        :param args: Array of InlineKeyboardButton to append to the keyboard
        :type args: :obj:`list` of
:class:`telebot.types.InlineKeyboardButton`

        :return: self, to allow function chaining.
        :rtype: :class:`telebot.types.InlineKeyboardMarkup`
        """

        return self.add(*args, row_width=self.max_row_keys)

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = dict()
        json_dict['inline_keyboard'] = [[button.to_dict() for button in row]
for row in self.keyboard]
        return json_dict


class InlineKeyboardButton(Dictionaryable, JsonSerializable,
JsonDeserializable):
    """
    This object represents one button of an inline keyboard. You must use
exactly one of the optional fields.

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

    :param text: Label text on the button
    :type text: :obj:`str`

    :param url: Optional. HTTP or tg:// URL to be opened when the button is
pressed. Links tg://user?id=<user_id> can be
        used to mention a user by their ID without using a username, if this
is allowed by their privacy settings.
    :type url: :obj:`str`

    :param callback_data: Optional. Data to be sent in a callback query to
the bot when button is pressed, 1-64 bytes
    :type callback_data: :obj:`str`
```

```
    :param web app: Optional. Description of the Web App that will be
launched when the user presses the button. The Web
        App will be able to send an arbitrary message on behalf of the user
using the method answerWebAppQuery. Available only
        in private chats between a user and the bot.
    :type web app: :class:`telebot.types.WebAppInfo`

    :param login_url: Optional. An HTTPS URL used to automatically authorize
the user. Can be used as a replacement for
        the Telegram Login Widget.
    :type login_url: :class:`telebot.types.LoginUrl`

    :param switch_inline_query: Optional. If set, pressing the button will
prompt the user to select one of their chats,
        open that chat and insert the bot's username and the specified inline
query in the input field. May be empty, in which
        case just the bot's username will be inserted.Note: This offers an
easy way for users to start using your bot in inline
        mode when they are currently in a private chat with it. Especially
useful when combined with switch_pm… actions - in
        this case the user will be automatically returned to the chat they
switched from, skipping the chat selection screen.
    :type switch inline query: :obj:`str`

    :param switch_inline_query_current_chat: Optional. If set, pressing the
button will insert the bot's username
        and the specified inline query in the current chat's input field. May
be empty, in which case only the bot's username
        will be inserted.This offers a quick way for the user to open your
bot in inline mode in the same chat - good for selecting
        something from multiple options.
    :type switch_inline_query_current_chat: :obj:`str`

    :param callback_game: Optional. Description of the game that will be
launched when the user presses the
        button. NOTE: This type of button must always be the first button in
the first row.
    :type callback_game: :class:`telebot.types.CallbackGame`

    :param pay: Optional. Specify True, to send a Pay button. NOTE: This type
of button must always be the first button in
        the first row and can only be used in invoice messages.
    :type pay: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineKeyboardButton`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'login_url' in obj:
            obj['login_url'] = LoginUrl.de_json(obj.get('login_url'))
        if 'web_app' in obj:
            obj['web app'] = WebAppInfo.de json(obj.get('web app'))

        return cls(**obj)

    def __init__(self, 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):
        self.text: str = text
```

```
        self.url: str = url
        self.callback_data: str = callback_data
        self.web_app: WebAppInfo = web_app
        self.switch_inline_query: str = switch_inline_query
        self.switch_inline_query_current_chat: str =
switch_inline_query_current_chat
        self.callback_game = callback_game # Not Implemented
        self.pay: bool = pay
        self.login_url: LoginUrl = login_url

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = {'text': self.text}
        if self.url:
            json_dict['url'] = self.url
        if self.callback_data:
            json_dict['callback_data'] = self.callback_data
        if self.web_app:
            json_dict['web_app'] = self.web_app.to_dict()
        if self.switch_inline_query is not None:
            json_dict['switch_inline_query'] = self.switch_inline_query
        if self.switch_inline_query_current_chat is not None:
            json_dict['switch_inline_query_current_chat'] =
self.switch_inline_query_current_chat
        if self.callback_game is not None:
            json_dict['callback_game'] = self.callback_game
        if self.pay is not None:
            json_dict['pay'] = self.pay
        if self.login_url is not None:
            json_dict['login_url'] = self.login_url.to_dict()
        return json_dict


class LoginUrl(Dictionaryable, JsonSerializable, JsonDeserializable):
    """
    This object represents a parameter of the inline keyboard button used to
automatically authorize a user. Serves as a great replacement for the
Telegram Login Widget when the user is coming from Telegram. All the user
needs to do is tap/click a button and confirm that they want to log in:

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

    :param url: An HTTPS URL to be opened with user authorization data added
to the query string when the button is pressed.
        If the user refuses to provide authorization data, the original URL
without information about the user will be
        opened. The data added is the same as described in Receiving
authorization data. NOTE: You must always check the hash
        of the received data to verify the authentication and the integrity
of the data as described in Checking
        authorization.
    :type url: :obj:`str`

    :param forward text: Optional. New text of the button in forwarded
messages.
    :type forward_text: :obj:`str`

    :param bot username: Optional. Username of a bot, which will be used for
user authorization. See Setting up a bot for
        more details. If not specified, the current bot's username will be
assumed. The url's domain must be the same as the
        domain linked with the bot. See Linking your domain to the bot for
```

```
more details.
    :type bot username: :obj:`str`

    :param request_write_access: Optional. Pass True to request the
permission for your bot to send messages to the
        user.
    :type request write access: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.LoginUrl`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, url, forward_text=None, bot_username=None,
request_write_access=None, **kwargs):
        self.url: str = url
        self.forward_text: str = forward_text
        self.bot_username: str = bot_username
        self.request_write_access: bool = request_write_access

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = {'url': self.url}
        if self.forward_text:
            json_dict['forward_text'] = self.forward_text
        if self.bot_username:
            json_dict['bot_username'] = self.bot_username
        if self.request_write_access is not None:
            json_dict['request_write_access'] = self.request_write_access
        return json_dict


class CallbackQuery(JsonDeserializable):
    """
    This object represents an incoming callback query from a callback button
in an inline keyboard. If the button that originated the query was attached
to a message sent by the bot, the field message will be present. If the
button was attached to a message sent via the bot (in inline mode), the field
inline_message_id will be present. Exactly one of the fields data or
game short name will be present.

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

    :param id: Unique identifier for this query
    :type id: :obj:`str`

    :param from_user: Sender
    :type from_user: :class:`telebot.types.User`

    :param message: Optional. Message with the callback button that
originated the query. Note that message content and
        message date will not be available if the message is too old
    :type message: :class:`telebot.types.Message`

    :param inline_message_id: Optional. Identifier of the message sent via
the bot in inline mode, that originated the
        query.
    :type inline_message_id: :obj:`str`
```

```
    :param chat instance: Global identifier, uniquely corresponding to the
chat to which the message with the callback
        button was sent. Useful for high scores in games.
    :type chat_instance: :obj:`str`

    :param data: Optional. Data associated with the callback button. Be aware
that the message originated the query can
        contain no callback buttons with this data.
    :type data: :obj:`str`

    :param game_short_name: Optional. Short name of a Game to be returned,
serves as the unique identifier for the game
    :type game_short_name: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.CallbackQuery`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if not "data" in obj:
            # "data" field is Optional in the API, but historically is
mandatory in the class constructor
            obj['data'] = None
        obj['from_user'] = User.de_json(obj.pop('from'))
        if 'message' in obj:
            obj['message'] = Message.de_json(obj.get('message'))
        obj['json_string'] = json_string
        return cls(**obj)

    def __init__(self, id, from_user, data, chat_instance, json_string,
message=None, inline_message_id=None, game_short_name=None, **kwargs):
        self.id: int = id
        self.from_user: User = from_user
        self.message: Message = message
        self.inline_message_id: str = inline_message_id
        self.chat_instance: str = chat_instance
        self.data: str = data
        self.game_short_name: str = game_short_name
        self.json = json_string


class ChatPhoto(JsonDeserializable):
    """
    This object represents a chat photo.

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

    :param small file id: File identifier of small (160x160) chat photo. This
file_id can be used only for photo
        download and only for as long as the photo is not changed.
    :type small_file_id: :obj:`str`

    :param small file unique id: Unique file identifier of small (160x160)
chat photo, which is supposed to be the same
        over time and for different bots. Can't be used to download or reuse
the file.
    :type small file unique id: :obj:`str`

    :param big_file_id: File identifier of big (640x640) chat photo. This
file_id can be used only for photo download and
        only for as long as the photo is not changed.
```

```
    :type big_file_id: :obj:`str`

    :param big_file_unique_id: Unique file identifier of big (640x640) chat
photo, which is supposed to be the same over
        time and for different bots. Can't be used to download or reuse the
file.
    :type big file unique id: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatPhoto`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, small_file_id, small_file_unique_id, big_file_id,
big_file_unique_id, **kwargs):
        self.small_file_id: str = small_file_id
        self.small_file_unique_id: str = small_file_unique_id
        self.big_file_id: str = big_file_id
        self.big_file_unique_id: str = big_file_unique_id


class ChatMember(JsonDeserializable):
    """
    This object contains information about one member of a chat.
    Currently, the following 6 types of chat members are supported:

    * :class:`telebot.types.ChatMemberOwner`
    * :class:`telebot.types.ChatMemberAdministrator`
    * :class:`telebot.types.ChatMemberMember`
    * :class:`telebot.types.ChatMemberRestricted`
    * :class:`telebot.types.ChatMemberLeft`
    * :class:`telebot.types.ChatMemberBanned`

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

    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        obj['user'] = User.de_json(obj['user'])
        member_type = obj['status']
        # Ordered according to estimated appearance frequency.
        if member_type == "member":
            return ChatMemberMember(**obj)
        elif member_type == "left":
            return ChatMemberLeft(**obj)
        elif member_type == "kicked":
            return ChatMemberBanned(**obj)
        elif member_type == "restricted":
            return ChatMemberRestricted(**obj)
        elif member_type == "administrator":
            return ChatMemberAdministrator(**obj)
        elif member_type == "creator":
            return ChatMemberOwner(**obj)
        else:
            # Should not be here. For "if something happen" compatibility
            return cls(**obj)

    def __init__(self, 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, can_manage_topics=None, **kwargs):
        self.user: User = user
        self.status: str = status
        self.custom_title: str = custom_title
        self.is_anonymous: bool = is_anonymous
        self.can_be_edited: bool = can_be_edited
        self.can_post_messages: bool = can_post_messages
        self.can_edit_messages: bool = can_edit_messages
        self.can_delete_messages: bool = can_delete_messages
        self.can_restrict_members: bool = can_restrict_members
        self.can_promote_members: bool = can_promote_members
        self.can_change_info: bool = can_change_info
        self.can_invite_users: bool = can_invite_users
        self.can_pin_messages: bool = can_pin_messages
        self.is_member: bool = is_member
        self.can_send_messages: bool = can_send_messages
        self.can_send_media_messages: bool = can_send_media_messages
        self.can_send_polls: bool = can_send_polls
        self.can_send_other_messages: bool = can_send_other_messages
        self.can_add_web_page_previews: bool = can_add_web_page_previews
        self.can_manage_chat: bool = can_manage_chat
        self.can_manage_video_chats: bool = can_manage_video_chats
        self.can_manage_voice_chats: bool = self.can_manage_video_chats   #
deprecated, for backward compatibility
        self.until_date: int = until_date
        self.can_manage_topics: bool = can_manage_topics


class ChatMemberOwner(ChatMember):
    """
    Represents a chat member that owns the chat and has all administrator
privileges.

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

    :param status: The member's status in the chat, always “creator”
    :type status: :obj:`str`

    :param user: Information about the user
    :type user: :class:`telebot.types.User`

    :param is_anonymous: True, if the user's presence in the chat is hidden
    :type is_anonymous: :obj:`bool`

    :param custom_title: Optional. Custom title for this user
    :type custom_title: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatMemberOwner`
    """
    pass
```

```
class ChatMemberAdministrator(ChatMember):
    """
    Represents a chat member that has some additional privileges.

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

    :param status: The member's status in the chat, always “administrator”
    :type status: :obj:`str`

    :param user: Information about the user
    :type user: :class:`telebot.types.User`

    :param can be edited: True, if the bot is allowed to edit administrator
privileges of that user
    :type can_be_edited: :obj:`bool`

    :param is anonymous: True, if the user's presence in the chat is hidden
    :type is_anonymous: :obj:`bool`

    :param can_manage_chat: 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
    :type can_manage_chat: :obj:`bool`

    :param can_delete_messages: True, if the administrator can delete
messages of other users
    :type can_delete_messages: :obj:`bool`

    :param can_manage_video_chats: True, if the administrator can manage
video chats
    :type can_manage_video_chats: :obj:`bool`

    :param can restrict members: True, if the administrator can restrict, ban
or unban chat members
    :type can_restrict_members: :obj:`bool`

    :param can promote members: True, if the administrator can add new
administrators with a subset of their own
        privileges or demote administrators that he has promoted, directly or
indirectly (promoted by administrators that
        were appointed by the user)
    :type can promote members: :obj:`bool`

    :param can_change_info: True, if the user is allowed to change the chat
title, photo and other settings
    :type can change info: :obj:`bool`

    :param can_invite_users: True, if the user is allowed to invite new users
to the chat
    :type can_invite_users: :obj:`bool`

    :param can_post_messages: Optional. True, if the administrator can post
in the channel; channels only
    :type can_post_messages: :obj:`bool`

    :param can_edit_messages: Optional. True, if the administrator can edit
messages of other users and can pin
        messages; channels only
    :type can_edit_messages: :obj:`bool`
```

```
    :param can pin messages: Optional. True, if the user is allowed to pin
messages; groups and supergroups only
    :type can_pin_messages: :obj:`bool`

    :param can manage topics: Optional. True, if the user is allowed to
create, rename, close, and reopen forum topics;
        supergroups only
    :type can_manage_topics: :obj:`bool`

    :param custom title: Optional. Custom title for this user
    :type custom_title: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatMemberAdministrator`
    """
    pass


class ChatMemberMember(ChatMember):
    """
    Represents a chat member that has no additional privileges or
restrictions.

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

    :param status: The member's status in the chat, always “member”
    :type status: :obj:`str`

    :param user: Information about the user
    :type user: :class:`telebot.types.User`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatMemberMember`
    """
    pass


class ChatMemberRestricted(ChatMember):
    """
    Represents a chat member that is under certain restrictions in the chat.
Supergroups only.

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

    :param status: The member's status in the chat, always “restricted”
    :type status: :obj:`str`

    :param user: Information about the user
    :type user: :class:`telebot.types.User`

    :param is_member: True, if the user is a member of the chat at the moment
of the request
    :type is member: :obj:`bool`

    :param can_change_info: True, if the user is allowed to change the chat
title, photo and other settings
    :type can change info: :obj:`bool`

    :param can_invite_users: True, if the user is allowed to invite new users
to the chat
    :type can_invite_users: :obj:`bool`
```

```
    :param can_pin_messages: True, if the user is allowed to pin messages
    :type can_pin_messages: :obj:`bool`

    :param can_manage_topics: True, if the user is allowed to create forum
topics
    :type can_manage_topics: :obj:`bool`

    :param can_send_messages: True, if the user is allowed to send text
messages, contacts, locations and venues
    :type can_send_messages: :obj:`bool`

    :param can_send_media_messages: True, if the user is allowed to send
audios, documents, photos, videos, video
        notes and voice notes
    :type can_send_media_messages: :obj:`bool`

    :param can_send_polls: True, if the user is allowed to send polls
    :type can_send_polls: :obj:`bool`

    :param can_send_other_messages: True, if the user is allowed to send
animations, games, stickers and use inline
        bots
    :type can_send_other_messages: :obj:`bool`

    :param can_add_web_page_previews: True, if the user is allowed to add web
page previews to their messages
    :type can_add_web_page_previews: :obj:`bool`

    :param until_date: Date when restrictions will be lifted for this user;
unix time. If 0, then the user is restricted
        forever
    :type until_date: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatMemberRestricted`
    """
    pass


class ChatMemberLeft(ChatMember):
    """
    Represents a chat member that isn't currently a member of the chat, but
may join it themselves.

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

    :param status: The member's status in the chat, always “left”
    :type status: :obj:`str`

    :param user: Information about the user
    :type user: :class:`telebot.types.User`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatMemberLeft`
    """
    pass


class ChatMemberBanned(ChatMember):
    """
    Represents a chat member that was banned in the chat and can't return to
the chat or view chat messages.
```

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

    :param status: The member's status in the chat, always “kicked”
    :type status: :obj:`str`

    :param user: Information about the user
    :type user: :class:`telebot.types.User`

    :param until_date: Date when restrictions will be lifted for this user;
unix time. If 0, then the user is banned
        forever
    :type until_date: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatMemberBanned`
    """
    pass


class ChatPermissions(JsonDeserializable, JsonSerializable, Dictionaryable):
    """
    Describes actions that a non-administrator user is allowed to take in a
chat.

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

    :param can send messages: Optional. True, if the user is allowed to send
text messages, contacts, locations and
        venues
    :type can_send_messages: :obj:`bool`

    :param can_send_media_messages: Optional. True, if the user is allowed to
send audios, documents, photos, videos,
        video notes and voice notes, implies can_send_messages
    :type can_send_media_messages: :obj:`bool`

    :param can_send_polls: Optional. True, if the user is allowed to send
polls, implies can_send_messages
    :type can_send_polls: :obj:`bool`

    :param can_send_other_messages: Optional. True, if the user is allowed to
send animations, games, stickers and use
        inline bots, implies can_send_media_messages
    :type can send other messages: :obj:`bool`

    :param can_add_web_page_previews: Optional. True, if the user is allowed
to add web page previews to their
        messages, implies can_send_media_messages
    :type can add web page previews: :obj:`bool`

    :param can_change_info: Optional. True, if the user is allowed to change
the chat title, photo and other settings.
        Ignored in public supergroups
    :type can change info: :obj:`bool`

    :param can_invite_users: Optional. True, if the user is allowed to invite
new users to the chat
    :type can invite users: :obj:`bool`

    :param can_pin_messages: Optional. True, if the user is allowed to pin
messages. Ignored in public supergroups
    :type can_pin_messages: :obj:`bool`
```

```
    :param can_manage_topics: Optional. True, if the user is allowed to
create forum topics. If omitted defaults to the
        value of can_pin_messages
    :type can_manage_topics: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatPermissions`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return json_string
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, 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,
                 can_manage_topics=None, **kwargs):
        self.can_send_messages: bool = can_send_messages
        self.can_send_media_messages: bool = can_send_media_messages
        self.can_send_polls: bool = can_send_polls
        self.can_send_other_messages: bool = can_send_other_messages
        self.can_add_web_page_previews: bool = can_add_web_page_previews
        self.can_change_info: bool = can_change_info
        self.can_invite_users: bool = can_invite_users
        self.can_pin_messages: bool = can_pin_messages
        self.can_manage_topics: bool = can_manage_topics

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = dict()
        if self.can_send_messages is not None:
            json_dict['can_send_messages'] = self.can_send_messages
        if self.can_send_media_messages is not None:
            json_dict['can_send_media_messages'] =
self.can_send_media_messages
        if self.can_send_polls is not None:
            json_dict['can_send_polls'] = self.can_send_polls
        if self.can_send_other_messages is not None:
            json_dict['can_send_other_messages'] =
self.can_send_other_messages
        if self.can_add_web_page_previews is not None:
            json_dict['can_add_web_page_previews'] =
self.can_add_web_page_previews
        if self.can_change_info is not None:
            json_dict['can_change_info'] = self.can_change_info
        if self.can_invite_users is not None:
            json_dict['can_invite_users'] = self.can_invite_users
        if self.can_pin_messages is not None:
            json_dict['can_pin_messages'] = self.can_pin_messages
        if self.can_manage_topics is not None:
            json dict['can manage topics'] = self.can manage topics

        return json_dict


class BotCommand(JsonSerializable, JsonDeserializable, Dictionaryable):
    """
    This object represents a bot command.
```

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

    :param command: Text of the command; 1-32 characters. Can contain only
lowercase English letters, digits and
        underscores.
    :type command: :obj:`str`

    :param description: Description of the command; 1-256 characters.
    :type description: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommand`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, command, description):
        self.command: str = command
        self.description: str = description

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {'command': self.command, 'description': self.description}


# BotCommandScopes

class BotCommandScope(ABC, JsonSerializable):
    """
    This object represents the scope to which bot commands are applied.
Currently, the following 7 scopes are supported:

    * :class:`BotCommandScopeDefault`
    * :class:`BotCommandScopeAllPrivateChats`
    * :class:`BotCommandScopeAllGroupChats`
    * :class:`BotCommandScopeAllChatAdministrators`
    * :class:`BotCommandScopeChat`
    * :class:`BotCommandScopeChatAdministrators`
    * :class:`BotCommandScopeChatMember`

    Determining list of commands
    The following algorithm is used to determine the list of commands for a
particular user viewing the bot menu. The first list of commands which is set
is returned:

    Commands in the chat with the bot:

    * :class:`BotCommandScopeChat` + language_code
    * :class:`BotCommandScopeChat`
    * :class:`BotCommandScopeAllPrivateChats` + language_code
    * :class:`BotCommandScopeAllPrivateChats`
    * :class:`BotCommandScopeDefault` + language_code
    * :class:`BotCommandScopeDefault`

    Commands in group and supergroup chats:

    * :class:`BotCommandScopeChatMember` + language_code
    * :class:`BotCommandScopeChatMember`
    * :class:`BotCommandScopeChatAdministrators` + language_code
```

```
(administrators only)
    * :class:`BotCommandScopeChatAdministrators` (administrators only)
    * :class:`BotCommandScopeChat` + language_code
    * :class:`BotCommandScopeChat`
    * :class:`BotCommandScopeAllChatAdministrators` + language_code
(administrators only)
    * :class:`BotCommandScopeAllChatAdministrators` (administrators only)
    * :class:`BotCommandScopeAllGroupChats` + language_code
    * :class:`BotCommandScopeAllGroupChats`
    * :class:`BotCommandScopeDefault` + language_code
    * :class:`BotCommandScopeDefault`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScope`
    """
    def __init__(self, type='default', chat_id=None, user_id=None):
        self.type: str = type
        self.chat_id: Optional[Union[int, str]] = chat_id
        self.user_id: Optional[Union[int, str]] = user_id

    def to_json(self):
        json_dict = {'type': self.type}
        if self.chat_id:
            json_dict['chat_id'] = self.chat_id
        if self.user_id:
            json_dict['user_id'] = self.user_id
        return json.dumps(json_dict)


class BotCommandScopeDefault(BotCommandScope):
    """
    Represents the default scope of bot commands. Default commands are used
if no commands with a narrower scope are specified for the user.

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

    :param type: Scope type, must be default
    :type type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScopeDefault`
    """
    def __init__(self):
        """
        Represents the default scope of bot commands.
        Default commands are used if no commands with a narrower scope are
specified for the user.
        """
        super(BotCommandScopeDefault, self).__init__(type='default')


class BotCommandScopeAllPrivateChats(BotCommandScope):
    """
    Represents the scope of bot commands, covering all private chats.

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

    :param type: Scope type, must be all private chats
    :type type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScopeAllPrivateChats`
```

```
    """
    def __init__(self):
        """
        Represents the scope of bot commands, covering all private chats.
        """
        super(BotCommandScopeAllPrivateChats,
self).__init__(type='all_private_chats')


class BotCommandScopeAllGroupChats(BotCommandScope):
    """
    Represents the scope of bot commands, covering all group and supergroup
chats.

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

    :param type: Scope type, must be all_group_chats
    :type type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScopeAllGroupChats`
    """
    def __init__(self):
        """
        Represents the scope of bot commands, covering all group and
supergroup chats.
        """
        super(BotCommandScopeAllGroupChats,
self).__init__(type='all_group_chats')


class BotCommandScopeAllChatAdministrators(BotCommandScope):
    """
    Represents the scope of bot commands, covering all group and supergroup
chat administrators.

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

    :param type: Scope type, must be all_chat_administrators
    :type type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScopeAllChatAdministrators`
    """
    def __init__(self):
        """
        Represents the scope of bot commands, covering all group and
supergroup chat administrators.
        """
        super(BotCommandScopeAllChatAdministrators,
self).__init__(type='all_chat_administrators')


class BotCommandScopeChat(BotCommandScope):
    """
    Represents the scope of bot commands, covering a specific chat.

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

    :param type: Scope type, must be chat
    :type type: :obj:`str`
```

```
    :param chat_id: Unique identifier for the target chat or username of the
target supergroup (in the format
        @supergroupusername)
    :type chat_id: :obj:`int` or :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScopeChat`
    """
    def __init__(self, chat_id=None):
        super(BotCommandScopeChat, self).__init__(type='chat',
chat_id=chat_id)


class BotCommandScopeChatAdministrators(BotCommandScope):
    """
    Represents the scope of bot commands, covering all administrators of a
specific group or supergroup chat.

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

    :param type: Scope type, must be chat_administrators
    :type type: :obj:`str`

    :param chat_id: Unique identifier for the target chat or username of the
target supergroup (in the format
        @supergroupusername)
    :type chat_id: :obj:`int` or :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScopeChatAdministrators`
    """
    def __init__(self, chat_id=None):
        super(BotCommandScopeChatAdministrators,
self).__init__(type='chat_administrators', chat_id=chat_id)


class BotCommandScopeChatMember(BotCommandScope):
    """
    Represents the scope of bot commands, covering a specific member of a
group or supergroup chat.

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

    :param type: Scope type, must be chat_member
    :type type: :obj:`str`

    :param chat_id: Unique identifier for the target chat or username of the
target supergroup (in the format
        @supergroupusername)
    :type chat_id: :obj:`int` or :obj:`str`

    :param user_id: Unique identifier of the target user
    :type user_id: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.BotCommandScopeChatMember`
    """
    def __init__(self, chat_id=None, user_id=None):
        super(BotCommandScopeChatMember, self).__init__(type='chat_member',
chat_id=chat_id, user_id=user_id)
```

```
# InlineQuery

class InlineQuery(JsonDeserializable):
    """
    This object represents an incoming inline query. When the user sends an
empty query, your bot could return some default or trending results.

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

    :param id: Unique identifier for this query
    :type id: :obj:`str`

    :param from_user: Sender
    :type from_user: :class:`telebot.types.User`

    :param query: Text of the query (up to 256 characters)
    :type query: :obj:`str`

    :param offset: Offset of the results to be returned, can be controlled by
the bot
    :type offset: :obj:`str`

    :param chat type: Optional. Type of the chat from which the inline query
was sent. Can be either “sender” for a private
        chat with the inline query sender, “private”, “group”, “supergroup”,
or “channel”. The chat type should be always
        known for requests sent from official clients and most third-party
clients, unless the request was sent from a secret
        chat
    :type chat_type: :obj:`str`

    :param location: Optional. Sender location, only for bots that request
user location
    :type location: :class:`telebot.types.Location`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQuery`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        obj['from_user'] = User.de_json(obj.pop('from'))
        if 'location' in obj:
            obj['location'] = Location.de_json(obj['location'])
        return cls(**obj)

    def __init__(self, id, from_user, query, offset, chat_type=None,
location=None, **kwargs):
        self.id: int = id
        self.from_user: User = from_user
        self.query: str = query
        self.offset: str = offset
        self.chat_type: str = chat_type
        self.location: Location = location


class InputTextMessageContent(Dictionaryable):
    """
    Represents the content of a text message to be sent as the result of an
inline query.

    Telegram Documentation:
```

```
https://core.telegram.org/bots/api#inputtextmessagecontent

    :param message_text: Text of the message to be sent, 1-4096 characters
    :type message_text: :obj:`str`

    :param parse mode: Optional. Mode for parsing entities in the message
text. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param entities: Optional. List of special entities that appear in
message text, which can be specified instead of
        parse_mode
    :type entities: :obj:`list` of :class:`telebot.types.MessageEntity`

    :param disable web page preview: Optional. Disables link previews for
links in the sent message
    :type disable_web_page_preview: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputTextMessageContent`
    """
    def __init__(self, message_text, parse_mode=None, entities=None,
disable_web_page_preview=None):
        self.message_text: str = message_text
        self.parse_mode: str = parse_mode
        self.entities: List[MessageEntity] = entities
        self.disable_web_page_preview: bool = disable_web_page_preview

    def to_dict(self):
        json_dict = {'message_text': self.message_text}
        if self.parse_mode:
            json_dict['parse_mode'] = self.parse_mode
        if self.entities:
            json_dict['entities'] =
MessageEntity.to_list_of_dicts(self.entities)
        if self.disable_web_page_preview is not None:
            json_dict['disable_web_page_preview'] =
self.disable_web_page_preview
        return json_dict


class InputLocationMessageContent(Dictionaryable):
    """
    Represents the content of a location message to be sent as the result of
an inline query.

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

    :param latitude: Latitude of the location in degrees
    :type latitude: :obj:`float`

    :param longitude: Longitude of the location in degrees
    :type longitude: :obj:`float`

    :param horizontal_accuracy: Optional. The radius of uncertainty for the
location, measured in meters; 0-1500
    :type horizontal_accuracy: :obj:`float` number

    :param live_period: Optional. Period in seconds for which the location
can be updated, should be between 60 and
        86400.
    :type live_period: :obj:`int`
```

```
    :param heading: Optional. For live locations, a direction in which the
user is moving, in degrees. Must be between 1
        and 360 if specified.
    :type heading: :obj:`int`

    :param proximity alert radius: Optional. For live locations, a maximum
distance for proximity alerts about
        approaching another chat member, in meters. Must be between 1 and
100000 if specified.
    :type proximity alert radius: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputLocationMessageContent`
    """
    def __init__(self, latitude, longitude, horizontal_accuracy=None,
live_period=None, heading=None, proximity_alert_radius=None):
        self.latitude: float = latitude
        self.longitude: float = longitude
        self.horizontal_accuracy: float = horizontal_accuracy
        self.live_period: int = live_period
        self.heading: int = heading
        self.proximity_alert_radius: int = proximity_alert_radius

    def to_dict(self):
        json_dict = {'latitude': self.latitude, 'longitude': self.longitude}
        if self.horizontal_accuracy:
            json_dict['horizontal_accuracy'] = self.horizontal_accuracy
        if self.live_period:
            json_dict['live_period'] = self.live_period
        if self.heading:
            json_dict['heading'] = self.heading
        if self.proximity_alert_radius:
            json_dict['proximity_alert_radius'] = self.proximity_alert_radius
        return json_dict


class InputVenueMessageContent(Dictionaryable):
    """
    Represents the content of a venue message to be sent as the result of an
inline query.

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

    :param latitude: Latitude of the venue in degrees
    :type latitude: :obj:`float`

    :param longitude: Longitude of the venue in degrees
    :type longitude: :obj:`float`

    :param title: Name of the venue
    :type title: :obj:`str`

    :param address: Address of the venue
    :type address: :obj:`str`

    :param foursquare_id: Optional. Foursquare identifier of the venue, if
known
    :type foursquare id: :obj:`str`

    :param foursquare_type: Optional. Foursquare type of the venue, if known.
(For example,
        “arts_entertainment/default”, “arts_entertainment/aquarium” or
```

```
“food/icecream”.)
    :type foursquare type: :obj:`str`

    :param google_place_id: Optional. Google Places identifier of the venue
    :type google_place_id: :obj:`str`

    :param google place type: Optional. Google Places type of the venue. (See
supported types.)
    :type google_place_type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputVenueMessageContent`
    """
    def __init__(self, latitude, longitude, title, address,
foursquare_id=None, foursquare_type=None,
                 google_place_id=None, google_place_type=None):
        self.latitude: float = latitude
        self.longitude: float = longitude
        self.title: str = title
        self.address: str = address
        self.foursquare_id: str = foursquare_id
        self.foursquare_type: str = foursquare_type
        self.google_place_id: str = google_place_id
        self.google_place_type: str = google_place_type

    def to_dict(self):
        json_dict = {
            'latitude': self.latitude,
            'longitude': self.longitude,
            'title': self.title,
            'address' : self.address
        }
        if self.foursquare_id:
            json_dict['foursquare_id'] = self.foursquare_id
        if self.foursquare_type:
            json_dict['foursquare_type'] = self.foursquare_type
        if self.google_place_id:
            json_dict['google_place_id'] = self.google_place_id
        if self.google_place_type:
            json_dict['google_place_type'] = self.google_place_type
        return json_dict


class InputContactMessageContent(Dictionaryable):
    """
    Represents the content of a contact message to be sent as the result of
an inline query.

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

    :param phone_number: Contact's phone number
    :type phone_number: :obj:`str`

    :param first_name: Contact's first name
    :type first name: :obj:`str`

    :param last_name: Optional. Contact's last name
    :type last_name: :obj:`str`

    :param vcard: Optional. Additional data about the contact in the form of
a vCard, 0-2048 bytes
    :type vcard: :obj:`str`
```

```
        :return: Instance of the class
        :rtype: :class:`telebot.types.InputContactMessageContent`
        """
        def __init__(self, phone_number, first_name, last_name=None, vcard=None):
            self.phone_number: str = phone_number
            self.first_name: str = first_name
            self.last_name: str = last_name
            self.vcard: str = vcard

        def to_dict(self):
            json_dict = {'phone_number': self.phone_number, 'first_name':
self.first_name}
            if self.last_name:
                json_dict['last_name'] = self.last_name
            if self.vcard:
                json_dict['vcard'] = self.vcard
            return json_dict


class InputInvoiceMessageContent(Dictionaryable):
        """
        Represents the content of an invoice message to be sent as the result of
an inline query.

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

        :param title: Product name, 1-32 characters
        :type title: :obj:`str`

        :param description: Product description, 1-255 characters
        :type description: :obj:`str`

        :param payload: Bot-defined invoice payload, 1-128 bytes. This will not
be displayed to the user, use for your
            internal processes.
        :type payload: :obj:`str`

        :param provider_token: Payment provider token, obtained via @BotFather
        :type provider_token: :obj:`str`

        :param currency: Three-letter ISO 4217 currency code, see more on
currencies
        :type currency: :obj:`str`

        :param prices: Price breakdown, a JSON-serialized list of components
(e.g. product price, tax, discount, delivery
            cost, delivery tax, bonus, etc.)
        :type prices: :obj:`list` of :class:`telebot.types.LabeledPrice`

        :param max tip amount: Optional. The maximum accepted amount for tips in
the smallest units of the currency
            (integer, not float/double). For example, for a maximum tip of US$
1.45 pass max_tip_amount = 145. See the exp
            parameter in currencies.json, it shows the number of digits past the
decimal point for each currency (2 for the
            majority of currencies). Defaults to 0
        :type max_tip_amount: :obj:`int`

        :param suggested tip amounts: Optional. A JSON-serialized array of
suggested amounts of tip in the smallest units
            of the currency (integer, not float/double). 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.
    :type suggested tip amounts: :obj:`list` of :obj:`int`

    :param provider_data: Optional. A JSON-serialized object for data about
the invoice, which will be shared with the
        payment provider. A detailed description of the required fields
should be provided by the payment provider.
    :type provider_data: :obj:`str`

    :param photo_url: Optional. URL of the product photo for the invoice. Can
be a photo of the goods or a marketing image
        for a service.
    :type photo_url: :obj:`str`

    :param photo_size: Optional. Photo size in bytes
    :type photo size: :obj:`int`

    :param photo_width: Optional. Photo width
    :type photo_width: :obj:`int`

    :param photo_height: Optional. Photo height
    :type photo_height: :obj:`int`

    :param need name: Optional. Pass True, if you require the user's full
name to complete the order
    :type need_name: :obj:`bool`

    :param need_phone_number: Optional. Pass True, if you require the user's
phone number to complete the order
    :type need_phone_number: :obj:`bool`

    :param need_email: Optional. Pass True, if you require the user's email
address to complete the order
    :type need_email: :obj:`bool`

    :param need_shipping_address: Optional. Pass True, if you require the
user's shipping address to complete the
        order
    :type need_shipping_address: :obj:`bool`

    :param send_phone_number_to_provider: Optional. Pass True, if the user's
phone number should be sent to provider
    :type send_phone_number_to_provider: :obj:`bool`

    :param send_email_to_provider: Optional. Pass True, if the user's email
address should be sent to provider
    :type send email to provider: :obj:`bool`

    :param is_flexible: Optional. Pass True, if the final price depends on
the shipping method
    :type is flexible: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputInvoiceMessageContent`
    """
    def   init  (self, 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):
        self.title: str = title
        self.description: str = description
        self.payload: str = payload
        self.provider_token: str = provider_token
        self.currency: str = currency
        self.prices: List[LabeledPrice] = prices
        self.max_tip_amount: Optional[int] = max_tip_amount
        self.suggested_tip_amounts: Optional[List[int]] =
suggested_tip_amounts
        self.provider_data: Optional[str] = provider_data
        self.photo_url: Optional[str] = photo_url
        self.photo_size: Optional[int] = photo_size
        self.photo_width: Optional[int] = photo_width
        self.photo_height: Optional[int] = photo_height
        self.need_name: Optional[bool] = need_name
        self.need_phone_number: Optional[bool] = need_phone_number
        self.need_email: Optional[bool] = need_email
        self.need_shipping_address: Optional[bool] = need_shipping_address
        self.send_phone_number_to_provider: Optional[bool] =
send_phone_number_to_provider
        self.send_email_to_provider: Optional[bool] = send_email_to_provider
        self.is_flexible: Optional[bool] = is_flexible

    def to_dict(self):
        json_dict = {
            'title': self.title,
            'description': self.description,
            'payload': self.payload,
            'provider_token': self.provider_token,
            'currency': self.currency,
            'prices': [LabeledPrice.to_dict(lp) for lp in self.prices]
        }
        if self.max_tip_amount:
            json_dict['max_tip_amount'] = self.max_tip_amount
        if self.suggested_tip_amounts:
            json_dict['suggested_tip_amounts'] = self.suggested_tip_amounts
        if self.provider_data:
            json_dict['provider_data'] = self.provider_data
        if self.photo_url:
            json_dict['photo_url'] = self.photo_url
        if self.photo_size:
            json_dict['photo_size'] = self.photo_size
        if self.photo_width:
            json_dict['photo_width'] = self.photo_width
        if self.photo_height:
            json_dict['photo_height'] = self.photo_height
        if self.need_name is not None:
            json_dict['need_name'] = self.need_name
        if self.need_phone_number is not None:
            json_dict['need_phone_number'] = self.need_phone_number
        if self.need_email is not None:
            json_dict['need_email'] = self.need_email
        if self.need_shipping_address is not None:
            json_dict['need_shipping_address'] = self.need_shipping_address
        if self.send phone number to provider is not None:
            json_dict['send_phone_number_to_provider'] =
self.send_phone_number_to_provider
        if self.send_email_to_provider is not None:
            json_dict['send_email_to_provider'] = self.send_email_to_provider
        if self.is_flexible is not None:
            json_dict['is_flexible'] = self.is_flexible
        return json_dict
```

```
class ChosenInlineResult(JsonDeserializable):
    """
    Represents a result of an inline query that was chosen by the user and
sent to their chat partner.

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

    :param result_id: The unique identifier for the result that was chosen
    :type result id: :obj:`str`

    :param from: The user that chose the result
    :type from: :class:`telebot.types.User`

    :param location: Optional. Sender location, only for bots that require
user location
    :type location: :class:`telebot.types.Location`

    :param inline message id: Optional. Identifier of the sent inline
message. Available only if there is an inline
        keyboard attached to the message. Will be also received in callback
queries and can be used to edit the message.
    :type inline message id: :obj:`str`

    :param query: The query that was used to obtain the result
    :type query: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChosenInlineResult`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        obj['from_user'] = User.de_json(obj.pop('from'))
        if 'location' in obj:
            obj['location'] = Location.de_json(obj['location'])
        return cls(**obj)

    def __init__(self, result_id, from_user, query, location=None,
inline_message_id=None, **kwargs):
        self.result_id: str = result_id
        self.from_user: User = from_user
        self.location: Location = location
        self.inline_message_id: str = inline_message_id
        self.query: str = query


class InlineQueryResultBase(ABC, Dictionaryable, JsonSerializable):
    """
    This object represents one result of an inline query. Telegram clients
currently support results of the following 20 types:

    * :class:`InlineQueryResultCachedAudio`
    * :class:`InlineQueryResultCachedDocument`
    * :class:`InlineQueryResultCachedGif`
    * :class:`InlineQueryResultCachedMpeg4Gif`
    * :class:`InlineQueryResultCachedPhoto`
    * :class:`InlineQueryResultCachedSticker`
    * :class:`InlineQueryResultCachedVideo`
    * :class:`InlineQueryResultCachedVoice`
    * :class:`InlineQueryResultArticle`
    * :class:`InlineQueryResultAudio`
```

```
    * :class:`InlineQueryResultContact`
    * :class:`InlineQueryResultGame`
    * :class:`InlineQueryResultDocument`
    * :class:`InlineQueryResultGif`
    * :class:`InlineQueryResultLocation`
    * :class:`InlineQueryResultMpeg4Gif`
    * :class:`InlineQueryResultPhoto`
    * :class:`InlineQueryResultVenue`
    * :class:`InlineQueryResultVideo`
    * :class:`InlineQueryResultVoice`

    Telegram Documentation:
https://core.telegram.org/bots/api#inlinequeryresult
    """
    # noinspection PyShadowingBuiltins
    def __init__(self, type, id, title = None, caption = None,
input_message_content = None,
                 reply_markup = None, caption_entities = None, parse_mode =
None):
        self.type = type
        self.id = id
        self.title = title
        self.caption = caption
        self.input_message_content = input_message_content
        self.reply_markup = reply_markup
        self.caption_entities = caption_entities
        self.parse_mode = parse_mode

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = {
            'type': self.type,
            'id': self.id
        }
        if self.title:
            json_dict['title'] = self.title
        if self.caption:
            json_dict['caption'] = self.caption
        if self.input_message_content:
            json_dict['input_message_content'] =
self.input_message_content.to_dict()
        if self.reply_markup:
            json_dict['reply_markup'] = self.reply_markup.to_dict()
        if self.caption_entities:
            json_dict['caption_entities'] =
MessageEntity.to_list_of_dicts(self.caption_entities)
        if self.parse_mode:
            json_dict['parse_mode'] = self.parse_mode
        return json_dict


class SentWebAppMessage(JsonDeserializable, Dictionaryable):
    """
    Describes an inline message sent by a Web App on behalf of a user.

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

    :param inline_message_id: Optional. Identifier of the sent inline
message. Available only if there is an inline
        keyboard attached to the message.
    :type inline_message_id: :obj:`str`
```

```
        :return: Instance of the class
        :rtype: :class:`telebot.types.SentWebAppMessage`
        """
        @classmethod
        def de_json(cls, json_string):
            if json_string is None: return None
            obj = cls.check_json(json_string)
            return cls(**obj)

        def __init__(self, inline_message_id=None):
            self.inline_message_id = inline_message_id

        def to_dict(self):
            json_dict = {}
            if self.inline_message_id:
                json_dict['inline_message_id'] = self.inline_message_id
            return json_dict


class InlineQueryResultArticle(InlineQueryResultBase):
        """
        Represents a link to an article or web page.

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

        :param type: Type of the result, must be article
        :type type: :obj:`str`

        :param id: Unique identifier for this result, 1-64 Bytes
        :type id: :obj:`str`

        :param title: Title of the result
        :type title: :obj:`str`

        :param input_message_content: Content of the message to be sent
        :type input message content: :class:`telebot.types.InputMessageContent`

        :param reply_markup: Optional. Inline keyboard attached to the message
        :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

        :param url: Optional. URL of the result
        :type url: :obj:`str`

        :param hide url: Optional. Pass True, if you don't want the URL to be
shown in the message
        :type hide_url: :obj:`bool`

        :param description: Optional. Short description of the result
        :type description: :obj:`str`

        :param thumb_url: Optional. Url of the thumbnail for the result
        :type thumb_url: :obj:`str`

        :param thumb width: Optional. Thumbnail width
        :type thumb_width: :obj:`int`

        :param thumb_height: Optional. Thumbnail height
        :type thumb height: :obj:`int`

        :return: Instance of the class
        :rtype: :class:`telebot.types.InlineQueryResultArticle`
        """
```

```
    def __init__(self, id, title, input_message_content, reply_markup=None,
                 url=None, hide_url=None, description=None, thumb_url=None,
thumb_width=None, thumb_height=None):
        super().__init__('article', id, title = title, input_message_content
= input_message_content, reply_markup = reply_markup)
        self.url = url
        self.hide_url = hide_url
        self.description = description
        self.thumb_url = thumb_url
        self.thumb_width = thumb_width
        self.thumb_height = thumb_height

    def to_dict(self):
        json_dict = super().to_dict()
        if self.url:
            json_dict['url'] = self.url
        if self.hide_url:
            json_dict['hide_url'] = self.hide_url
        if self.description:
            json_dict['description'] = self.description
        if self.thumb_url:
            json_dict['thumb_url'] = self.thumb_url
        if self.thumb_width:
            json_dict['thumb_width'] = self.thumb_width
        if self.thumb_height:
            json_dict['thumb_height'] = self.thumb_height
        return json_dict


class InlineQueryResultPhoto(InlineQueryResultBase):
    """
    Represents a link to a photo. By default, this photo will be sent by the
user with optional caption. Alternatively, you can use input_message_content
to send a message with the specified content instead of the photo.

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

    :param type: Type of the result, must be photo
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param photo_url: A valid URL of the photo. Photo must be in JPEG format.
Photo size must not exceed 5MB
    :type photo url: :obj:`str`

    :param thumb_url: URL of the thumbnail for the photo
    :type thumb_url: :obj:`str`

    :param photo_width: Optional. Width of the photo
    :type photo_width: :obj:`int`

    :param photo_height: Optional. Height of the photo
    :type photo height: :obj:`int`

    :param title: Optional. Title for the result
    :type title: :obj:`str`

    :param description: Optional. Short description of the result
    :type description: :obj:`str`

    :param caption: Optional. Caption of the photo to be sent, 0-1024
```

```
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the photo
caption. See formatting options for more
        details.
    :type parse mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the photo
    :type input message content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultPhoto`
    """
    def __init__(self, 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):
        super().__init__('photo', id, title = title, caption = caption,
                         input_message_content = input_message_content,
reply_markup = reply_markup,
                         parse_mode = parse_mode, caption_entities =
caption_entities)
        self.photo_url = photo_url
        self.thumb_url = thumb_url
        self.photo_width = photo_width
        self.photo_height = photo_height
        self.description = description

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['photo_url'] = self.photo_url
        json_dict['thumb_url'] = self.thumb_url
        if self.photo_width:
            json_dict['photo_width'] = self.photo_width
        if self.photo_height:
            json_dict['photo_height'] = self.photo_height
        if self.description:
            json_dict['description'] = self.description
        return json_dict


class InlineQueryResultGif(InlineQueryResultBase):
    """
    Represents a link to an animated GIF file. By default, this animated GIF
file will be sent by the user with optional caption. Alternatively, you can
use input_message_content to send a message with the specified content
instead of the animation.

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

    :param type: Type of the result, must be gif
    :type type: :obj:`str`
```

```
    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param gif_url: A valid URL for the GIF file. File size must not exceed
1MB
    :type gif url: :obj:`str`

    :param gif_width: Optional. Width of the GIF
    :type gif_width: :obj:`int`

    :param gif_height: Optional. Height of the GIF
    :type gif_height: :obj:`int`

    :param gif_duration: Optional. Duration of the GIF in seconds
    :type gif duration: :obj:`int`

    :param thumb_url: URL of the static (JPEG or GIF) or animated (MPEG4)
thumbnail for the result
    :type thumb url: :obj:`str`

    :param thumb_mime_type: Optional. MIME type of the thumbnail, must be one
of “image/jpeg”, “image/gif”, or
        “video/mp4”. Defaults to “image/jpeg”
    :type thumb_mime_type: :obj:`str`

    :param title: Optional. Title for the result
    :type title: :obj:`str`

    :param caption: Optional. Caption of the GIF file to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the caption.
See formatting options for more details.
    :type parse_mode: :obj:`str`

    :param caption entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input message content: Optional. Content of the message to be sent
instead of the GIF animation
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultGif`
    """
    def __init__(self, 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):
        super().__init__('gif', id, title = title, caption = caption,
                         input_message_content = input_message_content,
reply_markup = reply_markup,
                         parse_mode = parse_mode, caption_entities =
caption_entities)
```

```
        self.gif_url = gif_url
        self.gif_width = gif_width
        self.gif_height = gif_height
        self.thumb_url = thumb_url
        self.gif_duration = gif_duration
        self.thumb_mime_type = thumb_mime_type

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['gif_url'] = self.gif_url
        if self.gif_width:
            json_dict['gif_width'] = self.gif_width
        if self.gif_height:
            json_dict['gif_height'] = self.gif_height
        json_dict['thumb_url'] = self.thumb_url
        if self.gif_duration:
            json_dict['gif_duration'] = self.gif_duration
        if self.thumb_mime_type:
            json_dict['thumb_mime_type'] = self.thumb_mime_type
        return json_dict


class InlineQueryResultMpeg4Gif(InlineQueryResultBase):
    """
    Represents a link to a video animation (H.264/MPEG-4 AVC video without
sound). By default, this animated MPEG-4 file will be sent by the user with
optional caption. Alternatively, you can use input_message_content to send a
message with the specified content instead of the animation.

    Telegram Documentation:
https://core.telegram.org/bots/api#inlinequeryresultmpeg4gif

    :param type: Type of the result, must be mpeg4_gif
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param mpeg4_url: A valid URL for the MPEG4 file. File size must not
exceed 1MB
    :type mpeg4_url: :obj:`str`

    :param mpeg4_width: Optional. Video width
    :type mpeg4_width: :obj:`int`

    :param mpeg4 height: Optional. Video height
    :type mpeg4 height: :obj:`int`

    :param mpeg4_duration: Optional. Video duration in seconds
    :type mpeg4_duration: :obj:`int`

    :param thumb_url: URL of the static (JPEG or GIF) or animated (MPEG4)
thumbnail for the result
    :type thumb_url: :obj:`str`

    :param thumb mime type: Optional. MIME type of the thumbnail, must be one
of “image/jpeg”, “image/gif”, or
        “video/mp4”. Defaults to “image/jpeg”
    :type thumb_mime_type: :obj:`str`

    :param title: Optional. Title for the result
    :type title: :obj:`str`

    :param caption: Optional. Caption of the MPEG-4 file to be sent, 0-1024
```

```
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the caption.
See formatting options for more details.
    :type parse mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the video animation
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultMpeg4Gif`
    """
    def __init__(self, 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):
        super().__init__('mpeg4_gif', id, title = title, caption = caption,
                         input_message_content = input_message_content,
reply_markup = reply_markup,
                         parse_mode = parse_mode, caption_entities =
caption_entities)
        self.mpeg4_url = mpeg4_url
        self.mpeg4_width = mpeg4_width
        self.mpeg4_height = mpeg4_height
        self.thumb_url = thumb_url
        self.mpeg4_duration = mpeg4_duration
        self.thumb_mime_type = thumb_mime_type

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['mpeg4_url'] = self.mpeg4_url
        if self.mpeg4_width:
            json_dict['mpeg4_width'] = self.mpeg4_width
        if self.mpeg4_height:
            json_dict['mpeg4_height'] = self.mpeg4_height
        json_dict['thumb_url'] = self.thumb_url
        if self.mpeg4_duration:
            json_dict['mpeg4_duration '] = self.mpeg4_duration
        if self.thumb_mime_type:
            json_dict['thumb_mime_type'] = self.thumb_mime_type
        return json_dict


class InlineQueryResultVideo(InlineQueryResultBase):
    """
    Represents a link to a page containing an embedded video player or a
video file. By default, this video file will be sent by the user with an
optional caption. Alternatively, you can use input_message_content to send a
message with the specified content instead of the video.

    Telegram Documentation:
```

```
https://core.telegram.org/bots/api#inlinequeryresultvideo

    :param type: Type of the result, must be video
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param video_url: A valid URL for the embedded video player or video file
    :type video_url: :obj:`str`

    :param mime_type: MIME type of the content of the video URL, “text/html”
or “video/mp4”
    :type mime_type: :obj:`str`

    :param thumb url: URL of the thumbnail (JPEG only) for the video
    :type thumb_url: :obj:`str`

    :param title: Title for the result
    :type title: :obj:`str`

    :param caption: Optional. Caption of the video to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the video
caption. See formatting options for more
        details.
    :type parse mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param video_width: Optional. Video width
    :type video width: :obj:`int`

    :param video_height: Optional. Video height
    :type video_height: :obj:`int`

    :param video_duration: Optional. Video duration in seconds
    :type video_duration: :obj:`int`

    :param description: Optional. Short description of the result
    :type description: :obj:`str`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the video. This field is
        required if InlineQueryResultVideo is used to send an HTML-page as a
result (e.g., a YouTube video).
    :type input message content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultVideo`
    """
    def __init__(self, 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):
        super().__init__('video', id, title = title, caption = caption,
                         input_message_content = input_message_content,
reply_markup = reply_markup,
                         parse_mode = parse_mode, caption_entities =
caption_entities)
        self.video_url = video_url
        self.mime_type = mime_type
        self.thumb_url = thumb_url
        self.video_width = video_width
        self.video_height = video_height
        self.video_duration = video_duration
        self.description = description

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['video_url'] = self.video_url
        json_dict['mime_type'] = self.mime_type
        json_dict['thumb_url'] = self.thumb_url
        if self.video_height:
            json_dict['video_height'] = self.video_height
        if self.video_duration:
            json_dict['video_duration'] = self.video_duration
        if self.description:
            json_dict['description'] = self.description
        return json_dict


class InlineQueryResultAudio(InlineQueryResultBase):
    """
    Represents a link to an MP3 audio file. By default, this audio file will
be sent by the user. Alternatively, you can use input_message_content to send
a message with the specified content instead of the audio.

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

    :param type: Type of the result, must be audio
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param audio_url: A valid URL for the audio file
    :type audio_url: :obj:`str`

    :param title: Title
    :type title: :obj:`str`

    :param caption: Optional. Caption, 0-1024 characters after entities
parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the audio
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`
```

```
    :param performer: Optional. Performer
    :type performer: :obj:`str`

    :param audio_duration: Optional. Audio duration in seconds
    :type audio_duration: :obj:`int`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the audio
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultAudio`
    """
    def __init__(self, id, audio_url, title,
                 caption=None, caption_entities=None, parse_mode=None,
performer=None,
                 audio_duration=None, reply_markup=None,
input_message_content=None):
        super().__init__('audio', id, title = title, caption = caption,
                         input_message_content = input_message_content,
reply_markup = reply_markup,
                         parse_mode = parse_mode, caption_entities =
caption_entities)
        self.audio_url = audio_url
        self.performer = performer
        self.audio_duration = audio_duration

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['audio_url'] = self.audio_url
        if self.performer:
            json_dict['performer'] = self.performer
        if self.audio_duration:
            json_dict['audio_duration'] = self.audio_duration
        return json_dict


class InlineQueryResultVoice(InlineQueryResultBase):
    """
    Represents a link to a voice recording in an .OGG container encoded with
OPUS. By default, this voice recording will be sent by the user.
Alternatively, you can use input_message_content to send a message with the
specified content instead of the the voice message.

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

    :param type: Type of the result, must be voice
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param voice_url: A valid URL for the voice recording
    :type voice_url: :obj:`str`

    :param title: Recording title
    :type title: :obj:`str`

    :param caption: Optional. Caption, 0-1024 characters after entities
parsing
```

```
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the voice
message caption. See formatting options for
        more details.
    :type parse mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param voice_duration: Optional. Recording duration in seconds
    :type voice_duration: :obj:`int`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input message content: Optional. Content of the message to be sent
instead of the voice recording
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultVoice`
    """
    def __init__(self, id, voice_url, title, caption=None,
caption_entities=None,
                 parse_mode=None, voice_duration=None, reply_markup=None,
input_message_content=None):
        super().__init__('voice', id, title = title, caption = caption,
                         input_message_content = input_message_content,
reply_markup = reply_markup,
                         parse_mode = parse_mode, caption_entities =
caption_entities)
        self.voice_url = voice_url
        self.voice_duration = voice_duration

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['voice_url'] = self.voice_url
        if self.voice_duration:
            json_dict['voice_duration'] = self.voice_duration
        return json_dict


class InlineQueryResultDocument(InlineQueryResultBase):
    """
    Represents a link to a file. By default, this file will be sent by the
user with an optional caption. Alternatively, you can use
input message content to send a message with the specified content instead of
the file. Currently, only .PDF and .ZIP files can be sent using this method.

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

    :param type: Type of the result, must be document
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param title: Title for the result
    :type title: :obj:`str`
```

```
    :param caption: Optional. Caption of the document to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse mode: Optional. Mode for parsing entities in the document
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param document_url: A valid URL for the file
    :type document_url: :obj:`str`

    :param mime type: MIME type of the content of the file, either
“application/pdf” or “application/zip”
    :type mime_type: :obj:`str`

    :param description: Optional. Short description of the result
    :type description: :obj:`str`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the file
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :param thumb_url: Optional. URL of the thumbnail (JPEG only) for the file
    :type thumb_url: :obj:`str`

    :param thumb_width: Optional. Thumbnail width
    :type thumb width: :obj:`int`

    :param thumb_height: Optional. Thumbnail height
    :type thumb_height: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultDocument`
    """
    def __init__(self, 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):
        super().__init__('document', id, title = title, caption = caption,
                         input_message_content = input_message_content,
reply_markup = reply_markup,
                         parse_mode = parse_mode, caption_entities =
caption_entities)
        self.document url = document url
        self.mime_type = mime_type
        self.description = description
        self.thumb_url = thumb_url
        self.thumb_width = thumb_width
        self.thumb_height = thumb_height

    def to_dict(self):
        json_dict = super().to_dict()
```

```
        json_dict['document_url'] = self.document_url
        json_dict['mime_type'] = self.mime_type
        if self.description:
            json_dict['description'] = self.description
        if self.thumb_url:
            json_dict['thumb_url'] = self.thumb_url
        if self.thumb_width:
            json_dict['thumb_width'] = self.thumb_width
        if self.thumb_height:
            json_dict['thumb_height'] = self.thumb_height
        return json_dict


class InlineQueryResultLocation(InlineQueryResultBase):
    """
    Represents a location on a map. By default, the location will be sent by
the user. Alternatively, you can use input_message_content to send a message
with the specified content instead of the location.

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

    :param type: Type of the result, must be location
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 Bytes
    :type id: :obj:`str`

    :param latitude: Location latitude in degrees
    :type latitude: :obj:`float` number

    :param longitude: Location longitude in degrees
    :type longitude: :obj:`float` number

    :param title: Location title
    :type title: :obj:`str`

    :param horizontal accuracy: Optional. The radius of uncertainty for the
location, measured in meters; 0-1500
    :type horizontal_accuracy: :obj:`float` number

    :param live period: Optional. Period in seconds for which the location
can be updated, should be between 60 and
        86400.
    :type live_period: :obj:`int`

    :param heading: Optional. For live locations, a direction in which the
user is moving, in degrees. Must be between 1
        and 360 if specified.
    :type heading: :obj:`int`

    :param proximity_alert_radius: Optional. For live locations, a maximum
distance for proximity alerts about
        approaching another chat member, in meters. Must be between 1 and
100000 if specified.
    :type proximity alert radius: :obj:`int`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the location
    :type input_message_content: :class:`telebot.types.InputMessageContent`
```

```
    :param thumb_url: Optional. Url of the thumbnail for the result
    :type thumb url: :obj:`str`

    :param thumb_width: Optional. Thumbnail width
    :type thumb_width: :obj:`int`

    :param thumb height: Optional. Thumbnail height
    :type thumb_height: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultLocation`
    """
    def __init__(self, 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):
        super().__init__('location', id, title = title,
                         input_message_content = input_message_content,
reply_markup = reply_markup)
        self.latitude = latitude
        self.longitude = longitude
        self.horizontal_accuracy = horizontal_accuracy
        self.live_period = live_period
        self.heading: int = heading
        self.proximity_alert_radius: int = proximity_alert_radius
        self.thumb_url = thumb_url
        self.thumb_width = thumb_width
        self.thumb_height = thumb_height

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['latitude'] = self.latitude
        json_dict['longitude'] = self.longitude
        if self.horizontal_accuracy:
            json_dict['horizontal_accuracy'] = self.horizontal_accuracy
        if self.live_period:
            json_dict['live_period'] = self.live_period
        if self.heading:
            json_dict['heading'] = self.heading
        if self.proximity_alert_radius:
            json_dict['proximity_alert_radius'] = self.proximity_alert_radius
        if self.thumb_url:
            json_dict['thumb_url'] = self.thumb_url
        if self.thumb_width:
            json_dict['thumb_width'] = self.thumb_width
        if self.thumb_height:
            json_dict['thumb_height'] = self.thumb_height
        return json_dict


class InlineQueryResultVenue(InlineQueryResultBase):
    """
    Represents a venue. By default, the venue will be sent by the user.
Alternatively, you can use input_message_content to send a message with the
specified content instead of the venue.

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

    :param type: Type of the result, must be venue
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 Bytes
```

```
    :type id: :obj:`str`

    :param latitude: Latitude of the venue location in degrees
    :type latitude: :obj:`float`

    :param longitude: Longitude of the venue location in degrees
    :type longitude: :obj:`float`

    :param title: Title of the venue
    :type title: :obj:`str`

    :param address: Address of the venue
    :type address: :obj:`str`

    :param foursquare_id: Optional. Foursquare identifier of the venue if
known
    :type foursquare_id: :obj:`str`

    :param foursquare_type: Optional. Foursquare type of the venue, if known.
(For example,
        “arts_entertainment/default”, “arts_entertainment/aquarium” or
“food/icecream”.)
    :type foursquare_type: :obj:`str`

    :param google_place_id: Optional. Google Places identifier of the venue
    :type google_place_id: :obj:`str`

    :param google_place_type: Optional. Google Places type of the venue. (See
supported types.)
    :type google_place_type: :obj:`str`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the venue
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :param thumb_url: Optional. Url of the thumbnail for the result
    :type thumb_url: :obj:`str`

    :param thumb width: Optional. Thumbnail width
    :type thumb_width: :obj:`int`

    :param thumb_height: Optional. Thumbnail height
    :type thumb height: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultVenue`
    """
    def __init__(self, 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):
        super().__init__('venue', id, title = title,
                         input_message_content = input_message_content,
reply_markup = reply_markup)
        self.latitude = latitude
        self.longitude = longitude
        self.address = address
        self.foursquare_id = foursquare_id
        self.foursquare_type = foursquare_type
```

```
        self.google_place_id = google_place_id
        self.google_place_type = google_place_type
        self.thumb_url = thumb_url
        self.thumb_width = thumb_width
        self.thumb_height = thumb_height

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['latitude'] = self.latitude
        json_dict['longitude'] = self.longitude
        json_dict['address'] = self.address
        if self.foursquare_id:
            json_dict['foursquare_id'] = self.foursquare_id
        if self.foursquare_type:
            json_dict['foursquare_type'] = self.foursquare_type
        if self.google_place_id:
            json_dict['google_place_id'] = self.google_place_id
        if self.google_place_type:
            json_dict['google_place_type'] = self.google_place_type
        if self.thumb_url:
            json_dict['thumb_url'] = self.thumb_url
        if self.thumb_width:
            json_dict['thumb_width'] = self.thumb_width
        if self.thumb_height:
            json_dict['thumb_height'] = self.thumb_height
        return json_dict


class InlineQueryResultContact(InlineQueryResultBase):
    """
    Represents a contact with a phone number. By default, this contact will
be sent by the user. Alternatively, you can use input_message_content to send
a message with the specified content instead of the contact.

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

    :param type: Type of the result, must be contact
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 Bytes
    :type id: :obj:`str`

    :param phone_number: Contact's phone number
    :type phone_number: :obj:`str`

    :param first name: Contact's first name
    :type first_name: :obj:`str`

    :param last_name: Optional. Contact's last name
    :type last name: :obj:`str`

    :param vcard: Optional. Additional data about the contact in the form of
a vCard, 0-2048 bytes
    :type vcard: :obj:`str`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input message content: Optional. Content of the message to be sent
instead of the contact
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :param thumb_url: Optional. Url of the thumbnail for the result
```

```
    :type thumb_url: :obj:`str`

    :param thumb_width: Optional. Thumbnail width
    :type thumb_width: :obj:`int`

    :param thumb height: Optional. Thumbnail height
    :type thumb height: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultContact`
    """
    def __init__(self, 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):
        super().__init__('contact', id,
                         input_message_content = input_message_content,
reply_markup = reply_markup)
        self.phone_number = phone_number
        self.first_name = first_name
        self.last_name = last_name
        self.vcard = vcard
        self.thumb_url = thumb_url
        self.thumb_width = thumb_width
        self.thumb_height = thumb_height

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['phone_number'] = self.phone_number
        json_dict['first_name'] = self.first_name
        if self.last_name:
            json_dict['last_name'] = self.last_name
        if self.vcard:
            json_dict['vcard'] = self.vcard
        if self.thumb_url:
            json_dict['thumb_url'] = self.thumb_url
        if self.thumb_width:
            json_dict['thumb_width'] = self.thumb_width
        if self.thumb_height:
            json_dict['thumb_height'] = self.thumb_height
        return json_dict


class InlineQueryResultGame(InlineQueryResultBase):
    """
    Represents a Game.

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

    :param type: Type of the result, must be game
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param game_short_name: Short name of the game
    :type game_short_name: :obj:`str`

    :param reply markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultGame`
```

```
    """
    def __init__(self, id, game_short_name, reply_markup=None):
        super().__init__('game', id, reply_markup = reply_markup)
        self.game_short_name = game_short_name

    def to_dict(self):
        json_dict = super().to_dict()
        json_dict['game_short_name'] = self.game_short_name
        return json_dict


class InlineQueryResultCachedBase(ABC, JsonSerializable):
    """
    Base class of all InlineQueryResultCached* classes.
    """
    def __init__(self):
        self.type = None
        self.id = None
        self.title = None
        self.description = None
        self.caption = None
        self.reply_markup = None
        self.input_message_content = None
        self.parse_mode = None
        self.caption_entities = None
        self.payload_dic = {}

    def to_json(self):
        json_dict = self.payload_dic
        json_dict['type'] = self.type
        json_dict['id'] = self.id
        if self.title:
            json_dict['title'] = self.title
        if self.description:
            json_dict['description'] = self.description
        if self.caption:
            json_dict['caption'] = self.caption
        if self.reply_markup:
            json_dict['reply_markup'] = self.reply_markup.to_dict()
        if self.input_message_content:
            json_dict['input_message_content'] =
self.input_message_content.to_dict()
        if self.parse_mode:
            json_dict['parse_mode'] = self.parse_mode
        if self.caption_entities:
            json_dict['caption_entities'] =
MessageEntity.to_list_of_dicts(self.caption_entities)
        return json.dumps(json_dict)


class InlineQueryResultCachedPhoto(InlineQueryResultCachedBase):
    """
    Represents a link to a photo stored on the Telegram servers. By default,
this photo will be sent by the user with an optional caption. Alternatively,
you can use input_message_content to send a message with the specified
content instead of the photo.

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

    :param type: Type of the result, must be photo
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
```

```
    :type id: :obj:`str`

    :param photo_file_id: A valid file identifier of the photo
    :type photo_file_id: :obj:`str`

    :param title: Optional. Title for the result
    :type title: :obj:`str`

    :param description: Optional. Short description of the result
    :type description: :obj:`str`

    :param caption: Optional. Caption of the photo to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse mode: Optional. Mode for parsing entities in the photo
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the photo
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedPhoto`
    """
    def __init__(self, id, photo_file_id, title=None, description=None,
                 caption=None, caption_entities = None, parse_mode=None,
                 reply_markup=None, input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'photo'
        self.id = id
        self.photo_file_id = photo_file_id
        self.title = title
        self.description = description
        self.caption = caption
        self.caption_entities = caption_entities
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.parse_mode = parse_mode
        self.payload_dic['photo_file_id'] = photo_file_id


class InlineQueryResultCachedGif(InlineQueryResultCachedBase):
    """
    Represents a link to an animated GIF file stored on the Telegram servers.
By default, this animated GIF file will be sent by the user with an optional
caption. Alternatively, you can use input_message_content to send a message
with specified content instead of the animation.

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

    :param type: Type of the result, must be gif
```

```
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param gif file id: A valid file identifier for the GIF file
    :type gif file id: :obj:`str`

    :param title: Optional. Title for the result
    :type title: :obj:`str`

    :param caption: Optional. Caption of the GIF file to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse mode: Optional. Mode for parsing entities in the caption.
See formatting options for more details.
    :type parse_mode: :obj:`str`

    :param caption entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input message content: Optional. Content of the message to be sent
instead of the GIF animation
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedGif`
    """
    def __init__(self, id, gif_file_id, title=None, description=None,
                 caption=None, caption_entities = None, parse_mode=None,
                 reply_markup=None, input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'gif'
        self.id = id
        self.gif_file_id = gif_file_id
        self.title = title
        self.description = description
        self.caption = caption
        self.caption_entities = caption_entities
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.parse_mode = parse_mode
        self.payload_dic['gif_file_id'] = gif_file_id


class InlineQueryResultCachedMpeg4Gif(InlineQueryResultCachedBase):
    """
    Represents a link to a video animation (H.264/MPEG-4 AVC video without
sound) stored on the Telegram servers. By default, this animated MPEG-4 file
will be sent by the user with an optional caption. Alternatively, you can use
input_message_content to send a message with the specified content instead of
the animation.

    Telegram Documentation:
https://core.telegram.org/bots/api#inlinequeryresultcachedmpeg4gif

    :param type: Type of the result, must be mpeg4_gif
```

```
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param mpeg4 file id: A valid file identifier for the MPEG4 file
    :type mpeg4 file id: :obj:`str`

    :param title: Optional. Title for the result
    :type title: :obj:`str`

    :param caption: Optional. Caption of the MPEG-4 file to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse mode: Optional. Mode for parsing entities in the caption.
See formatting options for more details.
    :type parse_mode: :obj:`str`

    :param caption entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input message content: Optional. Content of the message to be sent
instead of the video animation
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedMpeg4Gif`
    """
    def __init__(self, id, mpeg4_file_id, title=None, description=None,
                 caption=None, caption_entities = None, parse_mode=None,
                 reply_markup=None, input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'mpeg4_gif'
        self.id = id
        self.mpeg4_file_id = mpeg4_file_id
        self.title = title
        self.description = description
        self.caption = caption
        self.caption_entities = caption_entities
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.parse_mode = parse_mode
        self.payload_dic['mpeg4_file_id'] = mpeg4_file_id


class InlineQueryResultCachedSticker(InlineQueryResultCachedBase):
    """
    Represents a link to a sticker stored on the Telegram servers. By
default, this sticker will be sent by the user. Alternatively, you can use
input_message_content to send a message with the specified content instead of
the sticker.

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

    :param type: Type of the result, must be sticker
    :type type: :obj:`str`
```

```
    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param sticker_file_id: A valid file identifier of the sticker
    :type sticker file id: :obj:`str`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input message content: Optional. Content of the message to be sent
instead of the sticker
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedSticker`
    """
    def __init__(self, id, sticker_file_id, reply_markup=None,
input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'sticker'
        self.id = id
        self.sticker_file_id = sticker_file_id
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.payload_dic['sticker_file_id'] = sticker_file_id


class InlineQueryResultCachedDocument(InlineQueryResultCachedBase):
    """
    Represents a link to a file stored on the Telegram servers. By default,
this file will be sent by the user with an optional caption. Alternatively,
you can use input_message_content to send a message with the specified
content instead of the file.

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

    :param type: Type of the result, must be document
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param title: Title for the result
    :type title: :obj:`str`

    :param document_file_id: A valid file identifier for the file
    :type document_file_id: :obj:`str`

    :param description: Optional. Short description of the result
    :type description: :obj:`str`

    :param caption: Optional. Caption of the document to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the document
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
```

```
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the file
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedDocument`
    """
    def __init__(self, id, document_file_id, title, description=None,
                 caption=None, caption_entities = None, parse_mode=None,
                 reply_markup=None, input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'document'
        self.id = id
        self.document_file_id = document_file_id
        self.title = title
        self.description = description
        self.caption = caption
        self.caption_entities = caption_entities
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.parse_mode = parse_mode
        self.payload_dic['document_file_id'] = document_file_id


class InlineQueryResultCachedVideo(InlineQueryResultCachedBase):
    """
    Represents a link to a video file stored on the Telegram servers. By
default, this video file will be sent by the user with an optional caption.
Alternatively, you can use input_message_content to send a message with the
specified content instead of the video.

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

    :param type: Type of the result, must be video
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param video_file_id: A valid file identifier for the video file
    :type video_file_id: :obj:`str`

    :param title: Title for the result
    :type title: :obj:`str`

    :param description: Optional. Short description of the result
    :type description: :obj:`str`

    :param caption: Optional. Caption of the video to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the video
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`
```

```
    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the video
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedVideo`
    """
    def __init__(self, id, video_file_id, title, description=None,
                 caption=None, caption_entities = None, parse_mode=None,
                 reply_markup=None,
                 input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'video'
        self.id = id
        self.video_file_id = video_file_id
        self.title = title
        self.description = description
        self.caption = caption
        self.caption_entities = caption_entities
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.parse_mode = parse_mode
        self.payload_dic['video_file_id'] = video_file_id


class InlineQueryResultCachedVoice(InlineQueryResultCachedBase):
    """
    Represents a link to a voice message stored on the Telegram servers. By
default, this voice message will be sent by the user. Alternatively, you can
use input_message_content to send a message with the specified content
instead of the voice message.

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

    :param type: Type of the result, must be voice
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param voice_file_id: A valid file identifier for the voice message
    :type voice_file_id: :obj:`str`

    :param title: Voice message title
    :type title: :obj:`str`

    :param caption: Optional. Caption, 0-1024 characters after entities
parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the voice
message caption. See formatting options for
        more details.
```

```
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply_markup: Optional. Inline keyboard attached to the message
    :type reply_markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the voice message
    :type input_message_content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedVoice`
    """
    def __init__(self, id, voice_file_id, title, caption=None,
caption_entities = None,
                 parse_mode=None, reply_markup=None,
input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'voice'
        self.id = id
        self.voice_file_id = voice_file_id
        self.title = title
        self.caption = caption
        self.caption_entities = caption_entities
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.parse_mode = parse_mode
        self.payload_dic['voice_file_id'] = voice_file_id


class InlineQueryResultCachedAudio(InlineQueryResultCachedBase):
    """
    Represents a link to an MP3 audio file stored on the Telegram servers. By
default, this audio file will be sent by the user. Alternatively, you can use
input_message_content to send a message with the specified content instead of
the audio.

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

    :param type: Type of the result, must be audio
    :type type: :obj:`str`

    :param id: Unique identifier for this result, 1-64 bytes
    :type id: :obj:`str`

    :param audio_file_id: A valid file identifier for the audio file
    :type audio_file_id: :obj:`str`

    :param caption: Optional. Caption, 0-1024 characters after entities
parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the audio
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
```

```
in the caption, which can be specified
        instead of parse mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param reply markup: Optional. Inline keyboard attached to the message
    :type reply markup: :class:`telebot.types.InlineKeyboardMarkup`

    :param input_message_content: Optional. Content of the message to be sent
instead of the audio
    :type input message content: :class:`telebot.types.InputMessageContent`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InlineQueryResultCachedAudio`
    """
    def __init__(self, id, audio_file_id, caption=None, caption_entities =
None,
                 parse_mode=None, reply_markup=None,
input_message_content=None):
        InlineQueryResultCachedBase.__init__(self)
        self.type = 'audio'
        self.id = id
        self.audio_file_id = audio_file_id
        self.caption = caption
        self.caption_entities = caption_entities
        self.reply_markup = reply_markup
        self.input_message_content = input_message_content
        self.parse_mode = parse_mode
        self.payload_dic['audio_file_id'] = audio_file_id


# Games

class Game(JsonDeserializable):
    """
    This object represents a game. Use BotFather to create and edit games,
their short names will act as unique identifiers.

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

    :param title: Title of the game
    :type title: :obj:`str`

    :param description: Description of the game
    :type description: :obj:`str`

    :param photo: Photo that will be displayed in the game message in chats.
    :type photo: :obj:`list` of :class:`telebot.types.PhotoSize`

    :param text: Optional. Brief description of the game or high scores
included in the game message. Can be
        automatically edited to include current high scores for the game when
the bot calls setGameScore, or manually edited
        using editMessageText. 0-4096 characters.
    :type text: :obj:`str`

    :param text_entities: Optional. Special entities that appear in text,
such as usernames, URLs, bot commands, etc.
    :type text_entities: :obj:`list` of :class:`telebot.types.MessageEntity`

    :param animation: Optional. Animation that will be displayed in the game
message in chats. Upload via BotFather
    :type animation: :class:`telebot.types.Animation`
```

```
    :return: Instance of the class
    :rtype: :class:`telebot.types.Game`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['photo'] = Game.parse_photo(obj['photo'])
        if 'text_entities' in obj:
            obj['text_entities'] = Game.parse_entities(obj['text_entities'])
        if 'animation' in obj:
            obj['animation'] = Animation.de_json(obj['animation'])
        return cls(**obj)

    @classmethod
    def parse_photo(cls, photo_size_array):
        """
        Parse the photo array into a list of PhotoSize objects
        """
        ret = []
        for ps in photo_size_array:
            ret.append(PhotoSize.de_json(ps))
        return ret

    @classmethod
    def parse_entities(cls, message_entity_array):
        """
        Parse the message entity array into a list of MessageEntity objects
        """
        ret = []
        for me in message_entity_array:
            ret.append(MessageEntity.de_json(me))
        return ret

    def __init__(self, title, description, photo, text=None,
text_entities=None, animation=None, **kwargs):
        self.title: str = title
        self.description: str = description
        self.photo: List[PhotoSize] = photo
        self.text: str = text
        self.text_entities: List[MessageEntity] = text_entities
        self.animation: Animation = animation


class Animation(JsonDeserializable):
    """
    This object represents an animation file (GIF or H.264/MPEG-4 AVC video
without sound).

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

    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`

    :param width: Video width as defined by sender
    :type width: :obj:`int`

    :param height: Video height as defined by sender
```

```
    :type height: :obj:`int`

    :param duration: Duration of the video in seconds as defined by sender
    :type duration: :obj:`int`

    :param thumb: Optional. Animation thumbnail as defined by sender
    :type thumb: :class:`telebot.types.PhotoSize`

    :param file_name: Optional. Original animation filename as defined by
sender
    :type file name: :obj:`str`

    :param mime_type: Optional. MIME type of the file as defined by sender
    :type mime_type: :obj:`str`

    :param file size: Optional. File size in bytes. It can be bigger than
2^31 and some programming languages may have
        difficulty/silent defects in interpreting it. But it has at most 52
significant bits, so a signed 64-bit integer or
        double-precision float type are safe for storing this value.
    :type file_size: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Animation`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        if 'thumb' in obj and 'file_id' in obj['thumb']:
            obj["thumb"] = PhotoSize.de_json(obj['thumb'])
        else:
            obj['thumb'] = None
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, width=None, height=None,
duration=None,
                 thumb=None, file_name=None, mime_type=None, file_size=None,
**kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.width: int = width
        self.height: int = height
        self.duration: int = duration
        self.thumb: PhotoSize = thumb
        self.file_name: str = file_name
        self.mime_type: str = mime_type
        self.file_size: int = file_size


class GameHighScore(JsonDeserializable):
    """
    This object represents one row of the high scores table for a game.

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

    :param position: Position in high score table for the game
    :type position: :obj:`int`

    :param user: User
    :type user: :class:`telebot.types.User`

    :param score: Score
    :type score: :obj:`int`
```

```
    :return: Instance of the class
    :rtype: :class:`telebot.types.GameHighScore`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['user'] = User.de_json(obj['user'])
        return cls(**obj)

    def __init__(self, position, user, score, **kwargs):
        self.position: int = position
        self.user: User = user
        self.score: int = score


# Payments

class LabeledPrice(JsonSerializable, Dictionaryable):
    """
    This object represents a portion of the price for goods or services.

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

    :param label: Portion label
    :type label: :obj:`str`

    :param amount: Price of the product in the smallest units of the currency
(integer, not float/double). For example,
        for a price of US$ 1.45 pass amount = 145. See the exp parameter in
currencies.json, it shows the number of digits past
        the decimal point for each currency (2 for the majority of
currencies).
    :type amount: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.LabeledPrice`
    """
    def __init__(self, label, amount):
        self.label: str = label
        self.amount: int = amount

    def to_dict(self):
        return {
            'label': self.label, 'amount': self.amount
        }

    def to_json(self):
        return json.dumps(self.to_dict())


class Invoice(JsonDeserializable):
    """
    This object contains basic information about an invoice.

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

    :param title: Product name
    :type title: :obj:`str`

    :param description: Product description
    :type description: :obj:`str`
```

```
    :param start_parameter: Unique bot deep-linking parameter that can be
used to generate this invoice
    :type start_parameter: :obj:`str`

    :param currency: Three-letter ISO 4217 currency code
    :type currency: :obj:`str`

    :param total_amount: Total price in the smallest units of the currency
(integer, not float/double). For example,
        for a price of US$ 1.45 pass amount = 145. See the exp parameter in
currencies.json, it shows the number of digits past
        the decimal point for each currency (2 for the majority of
currencies).
    :type total_amount: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Invoice`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, title, description, start_parameter, currency,
total_amount, **kwargs):
        self.title: str = title
        self.description: str = description
        self.start_parameter: str = start_parameter
        self.currency: str = currency
        self.total_amount: int = total_amount


class ShippingAddress(JsonDeserializable):
    """
    This object represents a shipping address.

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

    :param country_code: Two-letter ISO 3166-1 alpha-2 country code
    :type country code: :obj:`str`

    :param state: State, if applicable
    :type state: :obj:`str`

    :param city: City
    :type city: :obj:`str`

    :param street_line1: First line for the address
    :type street line1: :obj:`str`

    :param street_line2: Second line for the address
    :type street_line2: :obj:`str`

    :param post code: Address post code
    :type post_code: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ShippingAddress`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
```

```
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, country_code, state, city, street_line1, street_line2,
post_code, **kwargs):
        self.country_code: str = country_code
        self.state: str = state
        self.city: str = city
        self.street_line1: str = street_line1
        self.street_line2: str = street_line2
        self.post_code: str = post_code


class OrderInfo(JsonDeserializable):
    """
    This object represents information about an order.

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

    :param name: Optional. User name
    :type name: :obj:`str`

    :param phone_number: Optional. User's phone number
    :type phone number: :obj:`str`

    :param email: Optional. User email
    :type email: :obj:`str`

    :param shipping address: Optional. User shipping address
    :type shipping_address: :class:`telebot.types.ShippingAddress`

    :return: Instance of the class
    :rtype: :class:`telebot.types.OrderInfo`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['shipping_address'] =
ShippingAddress.de_json(obj.get('shipping_address'))
        return cls(**obj)

    def __init__(self, name=None, phone_number=None, email=None,
shipping_address=None, **kwargs):
        self.name: str = name
        self.phone_number: str = phone_number
        self.email: str = email
        self.shipping_address: ShippingAddress = shipping_address


class ShippingOption(JsonSerializable):
    """
    This object represents one shipping option.

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

    :param id: Shipping option identifier
    :type id: :obj:`str`

    :param title: Option title
    :type title: :obj:`str`

    :param prices: List of price portions
    :type prices: :obj:`list` of :class:`telebot.types.LabeledPrice`
```

```
    :return: Instance of the class
    :rtype: :class:`telebot.types.ShippingOption`
    """
    def __init__(self, id, title):
        self.id: str = id
        self.title: str = title
        self.prices: List[LabeledPrice] = []

    def add_price(self, *args):
        """
        Add LabeledPrice to ShippingOption

        :param args: LabeledPrices
        :type args: :obj:`LabeledPrice`

        :return: None
        """
        for price in args:
            self.prices.append(price)
        return self

    def to_json(self):
        price_list = []
        for p in self.prices:
            price_list.append(p.to_dict())
        json_dict = json.dumps({'id': self.id, 'title': self.title, 'prices':
price_list})
        return json_dict


class SuccessfulPayment(JsonDeserializable):
    """
    This object contains basic information about a successful payment.

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

    :param currency: Three-letter ISO 4217 currency code
    :type currency: :obj:`str`

    :param total amount: Total price in the smallest units of the currency
(integer, not float/double). For example,
        for a price of US$ 1.45 pass amount = 145. See the exp parameter in
currencies.json, it shows the number of digits past
        the decimal point for each currency (2 for the majority of
currencies).
    :type total_amount: :obj:`int`

    :param invoice_payload: Bot specified invoice payload
    :type invoice payload: :obj:`str`

    :param shipping_option_id: Optional. Identifier of the shipping option
chosen by the user
    :type shipping_option_id: :obj:`str`

    :param order_info: Optional. Order information provided by the user
    :type order_info: :class:`telebot.types.OrderInfo`

    :param telegram payment charge id: Telegram payment identifier
    :type telegram_payment_charge_id: :obj:`str`

    :param provider_payment_charge_id: Provider payment identifier
    :type provider_payment_charge_id: :obj:`str`
```

```
    :return: Instance of the class
    :rtype: :class:`telebot.types.SuccessfulPayment`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['order_info'] = OrderInfo.de_json(obj.get('order_info'))
        return cls(**obj)

    def __init__(self, currency, total_amount, invoice_payload,
shipping_option_id=None, order_info=None,
                 telegram_payment_charge_id=None,
provider_payment_charge_id=None, **kwargs):
        self.currency: str = currency
        self.total_amount: int = total_amount
        self.invoice_payload: str = invoice_payload
        self.shipping_option_id: str = shipping_option_id
        self.order_info: OrderInfo = order_info
        self.telegram_payment_charge_id: str = telegram_payment_charge_id
        self.provider_payment_charge_id: str = provider_payment_charge_id


class ShippingQuery(JsonDeserializable):
    """
    This object contains information about an incoming shipping query.

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

    :param id: Unique query identifier
    :type id: :obj:`str`

    :param from: User who sent the query
    :type from: :class:`telebot.types.User`

    :param invoice_payload: Bot specified invoice payload
    :type invoice payload: :obj:`str`

    :param shipping_address: User specified shipping address
    :type shipping_address: :class:`telebot.types.ShippingAddress`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ShippingQuery`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['from_user'] = User.de_json(obj.pop('from'))
        obj['shipping_address'] =
ShippingAddress.de_json(obj['shipping_address'])
        return cls(**obj)

    def __init__(self, id, from_user, invoice_payload, shipping_address,
**kwargs):
        self.id: str = id
        self.from_user: User = from_user
        self.invoice_payload: str = invoice_payload
        self.shipping_address: ShippingAddress = shipping_address


class PreCheckoutQuery(JsonDeserializable):
    """
```

```
    This object contains information about an incoming pre-checkout query.

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

    :param id: Unique query identifier
    :type id: :obj:`str`

    :param from: User who sent the query
    :type from: :class:`telebot.types.User`

    :param currency: Three-letter ISO 4217 currency code
    :type currency: :obj:`str`

    :param total_amount: Total price in the smallest units of the currency
(integer, not float/double). For example,
        for a price of US$ 1.45 pass amount = 145. See the exp parameter in
currencies.json, it shows the number of digits past
        the decimal point for each currency (2 for the majority of
currencies).
    :type total_amount: :obj:`int`

    :param invoice_payload: Bot specified invoice payload
    :type invoice payload: :obj:`str`

    :param shipping_option_id: Optional. Identifier of the shipping option
chosen by the user
    :type shipping_option_id: :obj:`str`

    :param order_info: Optional. Order information provided by the user
    :type order_info: :class:`telebot.types.OrderInfo`

    :return: Instance of the class
    :rtype: :class:`telebot.types.PreCheckoutQuery`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['from_user'] = User.de_json(obj.pop('from'))
        obj['order_info'] = OrderInfo.de_json(obj.get('order_info'))
        return cls(**obj)

    def __init__(self, id, from_user, currency, total_amount,
invoice_payload, shipping_option_id=None, order_info=None, **kwargs):
        self.id: str = id
        self.from_user: User = from_user
        self.currency: str = currency
        self.total_amount: int = total_amount
        self.invoice_payload: str = invoice_payload
        self.shipping_option_id: str = shipping_option_id
        self.order_info: OrderInfo = order_info


# Stickers

class StickerSet(JsonDeserializable):
    """
    This object represents a sticker set.

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

    :param name: Sticker set name
    :type name: :obj:`str`
```

```
    :param title: Sticker set title
    :type title: :obj:`str`

    :param sticker_type: Type of stickers in the set, currently one of
"regular", "mask", "custom emoji"
    :type sticker type: :obj:`str`

    :param is_animated: True, if the sticker set contains animated stickers
    :type is_animated: :obj:`bool`

    :param is_video: True, if the sticker set contains video stickers
    :type is_video: :obj:`bool`

    :param contains_masks: True, if the sticker set contains masks.
Deprecated since Bot API 6.2,
        use sticker_type instead.
    :type contains_masks: :obj:`bool`

    :param stickers: List of all set stickers
    :type stickers: :obj:`list` of :class:`telebot.types.Sticker`

    :param thumb: Optional. Sticker set thumbnail in the .WEBP, .TGS, or
.WEBM format
    :type thumb: :class:`telebot.types.PhotoSize`

    :return: Instance of the class
    :rtype: :class:`telebot.types.StickerSet`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        stickers = []
        for s in obj['stickers']:
            stickers.append(Sticker.de_json(s))
        obj['stickers'] = stickers
        if 'thumb' in obj and 'file_id' in obj['thumb']:
            obj['thumb'] = PhotoSize.de_json(obj['thumb'])
        else:
            obj['thumb'] = None
        return cls(**obj)

    def __init__(self, name, title, sticker_type, is_animated, is_video,
stickers, thumb=None, **kwargs):
        self.name: str = name
        self.title: str = title
        self.sticker_type: str = sticker_type
        self.is_animated: bool = is_animated
        self.is_video: bool = is_video
        self.stickers: List[Sticker] = stickers
        self.thumb: PhotoSize = thumb

    @property
    def contains_masks(self):
        """
        Deprecated since Bot API 6.2, use sticker_type instead.
        """
        logger.warning("contains_masks is deprecated, use sticker_type
instead")
        return self.sticker_type == 'mask'


class Sticker(JsonDeserializable):
```

```
    """
    This object represents a sticker.

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

    :param file_id: Identifier for this file, which can be used to download
or reuse the file
    :type file_id: :obj:`str`

    :param file_unique_id: Unique identifier for this file, which is supposed
to be the same over time and for different
        bots. Can't be used to download or reuse the file.
    :type file_unique_id: :obj:`str`

    :param type: Type of the sticker, currently one of “regular”, “mask”,
“custom_emoji”. The type of the sticker is
        independent from its format, which is determined by the fields
is_animated and is_video.
    :type type: :obj:`str`

    :param width: Sticker width
    :type width: :obj:`int`

    :param height: Sticker height
    :type height: :obj:`int`

    :param is_animated: True, if the sticker is animated
    :type is_animated: :obj:`bool`

    :param is_video: True, if the sticker is a video sticker
    :type is_video: :obj:`bool`

    :param thumb: Optional. Sticker thumbnail in the .WEBP or .JPG format
    :type thumb: :class:`telebot.types.PhotoSize`

    :param emoji: Optional. Emoji associated with the sticker
    :type emoji: :obj:`str`

    :param set_name: Optional. Name of the sticker set to which the sticker
belongs
    :type set_name: :obj:`str`

    :param premium_animation: Optional. Premium animation for the sticker, if
the sticker is premium
    :type premium_animation: :class:`telebot.types.File`

    :param mask position: Optional. For mask stickers, the position where the
mask should be placed
    :type mask_position: :class:`telebot.types.MaskPosition`

    :param custom emoji id: Optional. For custom emoji stickers, unique
identifier of the custom emoji
    :type custom_emoji_id: :obj:`str`

    :param file_size: Optional. File size in bytes
    :type file size: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Sticker`
    """

    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
```

```
        obj = cls.check_json(json_string)
        if 'thumb' in obj and 'file_id' in obj['thumb']:
            obj['thumb'] = PhotoSize.de_json(obj['thumb'])
        else:
            obj['thumb'] = None
        if 'mask_position' in obj:
            obj['mask_position'] = MaskPosition.de_json(obj['mask_position'])
        if 'premium_animation' in obj:
            obj['premium_animation'] = File.de_json(obj['premium_animation'])
        return cls(**obj)

    def __init__(self, file_id, file_unique_id, type, width, height,
is_animated,
                 is_video, thumb=None, emoji=None, set_name=None,
mask_position=None, file_size=None,
                 premium_animation=None, custom_emoji_id=None, **kwargs):
        self.file_id: str = file_id
        self.file_unique_id: str = file_unique_id
        self.type: str = type
        self.width: int = width
        self.height: int = height
        self.is_animated: bool = is_animated
        self.is_video: bool = is_video
        self.thumb: PhotoSize = thumb
        self.emoji: str = emoji
        self.set_name: str = set_name
        self.mask_position: MaskPosition = mask_position
        self.file_size: int = file_size
        self.premium_animation: File = premium_animation
        self.custom_emoji_id: int = custom_emoji_id



class MaskPosition(Dictionaryable, JsonDeserializable, JsonSerializable):
    """
    This object describes the position on faces where a mask should be placed
by default.

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

    :param point: The part of the face relative to which the mask should be
placed. One of “forehead”, “eyes”, “mouth”, or
        “chin”.
    :type point: :obj:`str`

    :param x_shift: Shift by X-axis measured in widths of the mask scaled to
the face size, from left to right. For example,
        choosing -1.0 will place mask just to the left of the default mask
position.
    :type x_shift: :obj:`float` number

    :param y_shift: Shift by Y-axis measured in heights of the mask scaled to
the face size, from top to bottom. For
        example, 1.0 will place the mask just below the default mask
position.
    :type y_shift: :obj:`float` number

    :param scale: Mask scaling coefficient. For example, 2.0 means double
size.
    :type scale: :obj:`float` number

    :return: Instance of the class
    :rtype: :class:`telebot.types.MaskPosition`
    """
```

```
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, point, x_shift, y_shift, scale, **kwargs):
        self.point: str = point
        self.x_shift: float = x_shift
        self.y_shift: float = y_shift
        self.scale: float = scale

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {'point': self.point, 'x_shift': self.x_shift, 'y_shift':
self.y_shift, 'scale': self.scale}


# InputMedia

class InputMedia(Dictionaryable, JsonSerializable):
    """
    This object represents the content of a media message to be sent. It
should be one of

    * :class:`InputMediaAnimation`
    * :class:`InputMediaDocument`
    * :class:`InputMediaAudio`
    * :class:`InputMediaPhoto`
    * :class:`InputMediaVideo`
    """
    def __init__(self, type, media, caption=None, parse_mode=None,
caption_entities=None):
        self.type: str = type
        self.media: str = media
        self.caption: Optional[str] = caption
        self.parse_mode: Optional[str] = parse_mode
        self.caption_entities: Optional[List[MessageEntity]] =
caption_entities

        if util.is_string(self.media):
            self._media_name = ''
            self._media_dic = self.media
        else:
            self._media_name = util.generate_random_token()
            self._media_dic = 'attach://{0}'.format(self._media_name)

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = {'type': self.type, 'media': self._media_dic}
        if self.caption:
            json_dict['caption'] = self.caption
        if self.parse_mode:
            json_dict['parse_mode'] = self.parse_mode
        if self.caption_entities:
            json_dict['caption_entities'] =
MessageEntity.to_list_of_dicts(self.caption_entities)
        return json_dict
```

```
    def convert_input_media(self):
        """
        :meta private:
        """
        if util.is_string(self.media):
            return self.to_json(), None

        return self.to_json(), {self._media_name: self.media}


class InputMediaPhoto(InputMedia):
    """
    Represents a photo to be sent.

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

    :param type: Type of the result, must be photo
    :type type: :obj:`str`

    :param media: File to send. Pass a file_id to send a file that exists on
the Telegram servers (recommended), pass an
        HTTP URL for Telegram to get a file from the Internet, or pass
"attach://<file attach name>" to upload a new one using
        multipart/form-data under <file_attach_name> name. More information
on Sending Files »
    :type media: :obj:`str`

    :param caption: Optional. Caption of the photo to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the photo
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputMediaPhoto`
    """
    def __init__(self, media, caption=None, parse_mode=None):
        if util.is_pil_image(media):
            media = util.pil_image_to_file(media)

        super(InputMediaPhoto, self).__init__(type="photo", media=media,
caption=caption, parse_mode=parse_mode)

    def to_dict(self):
        return super(InputMediaPhoto, self).to_dict()


class InputMediaVideo(InputMedia):
    """
    Represents a video to be sent.

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

```
    :param type: Type of the result, must be video
    :type type: :obj:`str`

    :param media: File to send. Pass a file_id to send a file that exists on
the Telegram servers (recommended), pass an
        HTTP URL for Telegram to get a file from the Internet, or pass
“attach://<file attach name>” to upload a new one using
        multipart/form-data under <file_attach_name> name. More information
on Sending Files »
    :type media: :obj:`str`

    :param thumb: Optional. 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>. More information on
Sending Files »
    :type thumb: InputFile or :obj:`str`

    :param caption: Optional. Caption of the video to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the video
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param width: Optional. Video width
    :type width: :obj:`int`

    :param height: Optional. Video height
    :type height: :obj:`int`

    :param duration: Optional. Video duration in seconds
    :type duration: :obj:`int`

    :param supports_streaming: Optional. Pass True, if the uploaded video is
suitable for streaming
    :type supports_streaming: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputMediaVideo`
    """
    def __init__(self, media, thumb=None, caption=None, parse_mode=None,
width=None, height=None, duration=None,
                 supports_streaming=None):
        super(InputMediaVideo, self).__init__(type="video", media=media,
caption=caption, parse_mode=parse_mode)
        self.thumb = thumb
        self.width = width
        self.height = height
        self.duration = duration
        self.supports_streaming = supports_streaming
```

```
    def to_dict(self):
        ret = super(InputMediaVideo, self).to_dict()
        if self.thumb:
            ret['thumb'] = self.thumb
        if self.width:
            ret['width'] = self.width
        if self.height:
            ret['height'] = self.height
        if self.duration:
            ret['duration'] = self.duration
        if self.supports_streaming:
            ret['supports_streaming'] = self.supports_streaming
        return ret


class InputMediaAnimation(InputMedia):
    """
    Represents an animation file (GIF or H.264/MPEG-4 AVC video without
sound) to be sent.

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

    :param type: Type of the result, must be animation
    :type type: :obj:`str`

    :param media: File to send. Pass a file_id to send a file that exists on
the Telegram servers (recommended), pass an
        HTTP URL for Telegram to get a file from the Internet, or pass
"attach://<file_attach_name>" to upload a new one using
        multipart/form-data under <file_attach_name> name. More information
on Sending Files »
    :type media: :obj:`str`

    :param thumb: Optional. 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>. More information on
Sending Files »
    :type thumb: InputFile or :obj:`str`

    :param caption: Optional. Caption of the animation to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the animation
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param width: Optional. Animation width
    :type width: :obj:`int`
```

```
    :param height: Optional. Animation height
    :type height: :obj:`int`

    :param duration: Optional. Animation duration in seconds
    :type duration: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputMediaAnimation`
    """
    def __init__(self, media, thumb=None, caption=None, parse_mode=None,
width=None, height=None, duration=None):
        super(InputMediaAnimation, self).__init__(type="animation",
media=media, caption=caption, parse_mode=parse_mode)
        self.thumb = thumb
        self.width = width
        self.height = height
        self.duration = duration

    def to_dict(self):
        ret = super(InputMediaAnimation, self).to_dict()
        if self.thumb:
            ret['thumb'] = self.thumb
        if self.width:
            ret['width'] = self.width
        if self.height:
            ret['height'] = self.height
        if self.duration:
            ret['duration'] = self.duration
        return ret


class InputMediaAudio(InputMedia):
    """
    Represents an audio file to be treated as music to be sent.

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

    :param type: Type of the result, must be audio
    :type type: :obj:`str`

    :param media: File to send. Pass a file_id to send a file that exists on
the Telegram servers (recommended), pass an
        HTTP URL for Telegram to get a file from the Internet, or pass
"attach://<file attach name>" to upload a new one using
        multipart/form-data under <file attach name> name. More information
on Sending Files »
    :type media: :obj:`str`

    :param thumb: Optional. 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>. More information on
Sending Files »
    :type thumb: InputFile or :obj:`str`

    :param caption: Optional. Caption of the audio to be sent, 0-1024
characters after entities parsing
```

```
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the audio
caption. See formatting options for more
        details.
    :type parse mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param duration: Optional. Duration of the audio in seconds
    :type duration: :obj:`int`

    :param performer: Optional. Performer of the audio
    :type performer: :obj:`str`

    :param title: Optional. Title of the audio
    :type title: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputMediaAudio`
    """
    def __init__(self, media, thumb=None, caption=None, parse_mode=None,
duration=None, performer=None, title=None):
        super(InputMediaAudio, self).__init__(type="audio", media=media,
caption=caption, parse_mode=parse_mode)
        self.thumb = thumb
        self.duration = duration
        self.performer = performer
        self.title = title

    def to_dict(self):
        ret = super(InputMediaAudio, self).to_dict()
        if self.thumb:
            ret['thumb'] = self.thumb
        if self.duration:
            ret['duration'] = self.duration
        if self.performer:
            ret['performer'] = self.performer
        if self.title:
            ret['title'] = self.title
        return ret


class InputMediaDocument(InputMedia):
    """
    Represents a general file to be sent.

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

    :param type: Type of the result, must be document
    :type type: :obj:`str`

    :param media: File to send. Pass a file_id to send a file that exists on
the Telegram servers (recommended), pass an
        HTTP URL for Telegram to get a file from the Internet, or pass
"attach://<file_attach_name>" to upload a new one using
        multipart/form-data under <file_attach_name> name. More information
on Sending Files »
    :type media: :obj:`str`
```

```
    :param thumb: Optional. 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>. More information on
Sending Files »
    :type thumb: InputFile or :obj:`str`

    :param caption: Optional. Caption of the document to be sent, 0-1024
characters after entities parsing
    :type caption: :obj:`str`

    :param parse_mode: Optional. Mode for parsing entities in the document
caption. See formatting options for more
        details.
    :type parse_mode: :obj:`str`

    :param caption_entities: Optional. List of special entities that appear
in the caption, which can be specified
        instead of parse_mode
    :type caption_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param disable content type detection: Optional. Disables automatic
server-side content type detection for
        files uploaded using multipart/form-data. Always True, if the
document is sent as part of an album.
    :type disable_content_type_detection: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.InputMediaDocument`
    """
    def __init__(self, media, thumb=None, caption=None, parse_mode=None,
disable_content_type_detection=None):
        super(InputMediaDocument, self).__init__(type="document",
media=media, caption=caption, parse_mode=parse_mode)
        self.thumb = thumb
        self.disable_content_type_detection = disable_content_type_detection

    def to_dict(self):
        ret = super(InputMediaDocument, self).to_dict()
        if self.thumb:
            ret['thumb'] = self.thumb
        if self.disable_content_type_detection is not None:
            ret['disable_content_type_detection'] =
self.disable_content_type_detection
        return ret


class PollOption(JsonDeserializable):
    """
    This object contains information about one answer option in a poll.

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

    :param text: Option text, 1-100 characters
    :type text: :obj:`str`

    :param voter_count: Number of users that voted for this option
```

```
    :type voter_count: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.PollOption`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, text, voter_count = 0, **kwargs):
        self.text: str = text
        self.voter_count: int = voter_count
    # Converted in _convert_poll_options
    # def to_json(self):
    #     # send_poll Option is a simple string:
https://core.telegram.org/bots/api#sendpoll
    #     return json.dumps(self.text)


class Poll(JsonDeserializable):
    """
    This object contains information about a poll.

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

    :param id: Unique poll identifier
    :type id: :obj:`str`

    :param question: Poll question, 1-300 characters
    :type question: :obj:`str`

    :param options: List of poll options
    :type options: :obj:`list` of :class:`telebot.types.PollOption`

    :param total_voter_count: Total number of users that voted in the poll
    :type total_voter_count: :obj:`int`

    :param is_closed: True, if the poll is closed
    :type is_closed: :obj:`bool`

    :param is_anonymous: True, if the poll is anonymous
    :type is_anonymous: :obj:`bool`

    :param type: Poll type, currently can be “regular” or “quiz”
    :type type: :obj:`str`

    :param allows_multiple_answers: True, if the poll allows multiple answers
    :type allows_multiple_answers: :obj:`bool`

    :param correct_option_id: Optional. 0-based identifier of the correct
answer option. Available only for polls in
        the quiz mode, which are closed, or was sent (not forwarded) by the
bot or to the private chat with the bot.
    :type correct_option_id: :obj:`int`

    :param explanation: Optional. Text that is shown when a user chooses an
incorrect answer or taps on the lamp icon in a
        quiz-style poll, 0-200 characters
    :type explanation: :obj:`str`

    :param explanation_entities: Optional. Special entities like usernames,
URLs, bot commands, etc. that appear in
```

```
        the explanation
    :type explanation_entities: :obj:`list` of
:class:`telebot.types.MessageEntity`

    :param open_period: Optional. Amount of time in seconds the poll will be
active after creation
    :type open_period: :obj:`int`

    :param close_date: Optional. Point in time (Unix timestamp) when the poll
will be automatically closed
    :type close_date: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.Poll`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['poll_id'] = obj.pop('id')
        options = []
        for opt in obj['options']:
            options.append(PollOption.de_json(opt))
        obj['options'] = options or None
        if 'explanation_entities' in obj:
            obj['explanation_entities'] =
Message.parse_entities(obj['explanation_entities'])
        return cls(**obj)

    # noinspection PyShadowingBuiltins
    def __init__(
            self,
            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):
        self.id: str = poll_id
        self.question: str = question
        self.options: List[PollOption] = options
        self.total_voter_count: int = total_voter_count
        self.is_closed: bool = is_closed
        self.is_anonymous: bool = is_anonymous
        self.type: str = type
        if poll_type is not None:
            # Wrong param name backward compatibility
            logger.warning("Poll: poll_type parameter is deprecated. Use type
instead.")
            if type is None:
                self.type: str = poll_type
        self.allows_multiple_answers: bool = allows_multiple_answers
        self.correct_option_id: int = correct_option_id
        self.explanation: str = explanation
        self.explanation_entities: List[MessageEntity] = explanation_entities
        self.open_period: int = open_period
        self.close_date: int = close_date

    def add(self, option):
        """
        Add an option to the poll.

        :param option: Option to add
        :type option: :class:`telebot.types.PollOption` or :obj:`str`
```

```
        """
        if type(option) is PollOption:
            self.options.append(option)
        else:
            self.options.append(PollOption(option))


class PollAnswer(JsonSerializable, JsonDeserializable, Dictionaryable):
    """
    This object represents an answer of a user in a non-anonymous poll.

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

    :param poll_id: Unique poll identifier
    :type poll_id: :obj:`str`

    :param user: The user, who changed the answer to the poll
    :type user: :class:`telebot.types.User`

    :param option_ids: 0-based identifiers of answer options, chosen by the
user. May be empty if the user retracted
        their vote.
    :type option_ids: :obj:`list` of :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.PollAnswer`
    """
    @classmethod
    def de_json(cls, json_string):
        if (json_string is None): return None
        obj = cls.check_json(json_string)
        obj['user'] = User.de_json(obj['user'])
        return cls(**obj)

    def __init__(self, poll_id, user, option_ids, **kwargs):
        self.poll_id: str = poll_id
        self.user: User = user
        self.option_ids: List[int] = option_ids

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {'poll_id': self.poll_id,
                'user': self.user.to_dict(),
                'option_ids': self.option_ids}


class ChatLocation(JsonSerializable, JsonDeserializable, Dictionaryable):
    """
    Represents a location to which a chat is connected.

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

    :param location: The location to which the supergroup is connected. Can't
be a live location.
    :type location: :class:`telebot.types.Location`

    :param address: Location address; 1-64 characters, as defined by the chat
owner
    :type address: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatLocation`
```

```
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return json_string
        obj = cls.check_json(json_string)
        obj['location'] = Location.de_json(obj['location'])
        return cls(**obj)

    def __init__(self, location, address, **kwargs):
        self.location: Location = location
        self.address: str = address

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        return {
            "location": self.location.to_dict(),
            "address": self.address
        }


class ChatInviteLink(JsonSerializable, JsonDeserializable, Dictionaryable):
    """
    Represents an invite link for a chat.

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

    :param invite_link: The invite link. If the link was created by another
chat administrator, then the second part of
        the link will be replaced with "…".
    :type invite_link: :obj:`str`

    :param creator: Creator of the link
    :type creator: :class:`telebot.types.User`

    :param creates_join_request: True, if users joining the chat via the link
need to be approved by chat administrators
    :type creates_join_request: :obj:`bool`

    :param is_primary: True, if the link is primary
    :type is_primary: :obj:`bool`

    :param is_revoked: True, if the link is revoked
    :type is_revoked: :obj:`bool`

    :param name: Optional. Invite link name
    :type name: :obj:`str`

    :param expire_date: Optional. Point in time (Unix timestamp) when the
link will expire or has been expired
    :type expire_date: :obj:`int`

    :param member_limit: Optional. The maximum number of users that can be
members of the chat simultaneously after
        joining the chat via this invite link; 1-99999
    :type member_limit: :obj:`int`

    :param pending_join_request_count: Optional. Number of pending join
requests created using this link
    :type pending_join_request_count: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatInviteLink`
```

```
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        obj['creator'] = User.de_json(obj['creator'])
        return cls(**obj)

    def __init__(self, invite_link, creator, creates_join_request ,
is_primary, is_revoked,
                 name=None, expire_date=None, member_limit=None,
pending_join_request_count=None, **kwargs):
        self.invite_link: str = invite_link
        self.creator: User = creator
        self.creates_join_request: bool = creates_join_request
        self.is_primary: bool = is_primary
        self.is_revoked: bool = is_revoked
        self.name: str = name
        self.expire_date: int = expire_date
        self.member_limit: int = member_limit
        self.pending_join_request_count: int = pending_join_request_count

    def to_json(self):
        return json.dumps(self.to_dict())

    def to_dict(self):
        json_dict = {
            "invite_link": self.invite_link,
            "creator": self.creator.to_dict(),
            "is_primary": self.is_primary,
            "is_revoked": self.is_revoked,
            "creates_join_request": self.creates_join_request
        }
        if self.expire_date:
            json_dict["expire_date"] = self.expire_date
        if self.member_limit:
            json_dict["member_limit"] = self.member_limit
        if self.pending_join_request_count:
            json_dict["pending_join_request_count"] =
self.pending_join_request_count
        if self.name:
            json_dict["name"] = self.name
        return json_dict


class ProximityAlertTriggered(JsonDeserializable):
    """
    This object represents the content of a service message, sent whenever a
user in the chat triggers a proximity alert set by another user.

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

    :param traveler: User that triggered the alert
    :type traveler: :class:`telebot.types.User`

    :param watcher: User that set the alert
    :type watcher: :class:`telebot.types.User`

    :param distance: The distance between the users
    :type distance: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ProximityAlertTriggered`
```

```
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, traveler, watcher, distance, **kwargs):
        self.traveler: User = traveler
        self.watcher: User = watcher
        self.distance: int = distance


class VideoChatStarted(JsonDeserializable):
    """
    This object represents a service message about a video chat started in
the chat. Currently holds no information.
    """
    @classmethod
    def de_json(cls, json_string):
        return cls()

    def __init__(self):
        pass

class VoiceChatStarted(VideoChatStarted):
    """
    Deprecated, use :class:`VideoChatStarted` instead.
    """

    def __init__(self):
        logger.warning('VoiceChatStarted is deprecated. Use VideoChatStarted
instead.')
        super().__init__()


class VideoChatScheduled(JsonDeserializable):
    """
    This object represents a service message about a video chat scheduled in
the chat.

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

    :param start_date: Point in time (Unix timestamp) when the video chat is
supposed to be started by a chat
        administrator
    :type start_date: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.VideoChatScheduled`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, start_date, **kwargs):
        self.start_date: int = start_date

class VoiceChatScheduled(VideoChatScheduled):
    """
    Deprecated, use :class:`VideoChatScheduled` instead.
```

```
    """
    def __init__(self, *args, **kwargs):
        logger.warning('VoiceChatScheduled is deprecated. Use
VideoChatScheduled instead.')
        super().__init__(*args, **kwargs)


class VideoChatEnded(JsonDeserializable):
    """
    This object represents a service message about a video chat ended in the
chat.

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

    :param duration: Video chat duration in seconds
    :type duration: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.VideoChatEnded`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, duration, **kwargs):
        self.duration: int = duration

class VoiceChatEnded(VideoChatEnded):
    """
    Deprecated, use :class:`VideoChatEnded` instead.
    """
    def __init__(self, *args, **kwargs):
        logger.warning('VoiceChatEnded is deprecated. Use VideoChatEnded
instead.')
        super().__init__(*args, **kwargs)



class VideoChatParticipantsInvited(JsonDeserializable):
    """
    This object represents a service message about new members invited to a
video chat.

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

    :param users: New members that were invited to the video chat
    :type users: :obj:`list` of :class:`telebot.types.User`

    :return: Instance of the class
    :rtype: :class:`telebot.types.VideoChatParticipantsInvited`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        if 'users' in obj:
            obj['users'] = [User.de_json(u) for u in obj['users']]
        return cls(**obj)

    def __init__(self, users=None, **kwargs):
        self.users: List[User] = users
```

```
class VoiceChatParticipantsInvited(VideoChatParticipantsInvited):
    """
    Deprecated, use :class:`VideoChatParticipantsInvited` instead.
    """
    def __init__(self, *args, **kwargs):
        logger.warning('VoiceChatParticipantsInvited is deprecated. Use
VideoChatParticipantsInvited instead.')
        super().__init__(*args, **kwargs)


class MessageAutoDeleteTimerChanged(JsonDeserializable):
    """
    This object represents a service message about a change in auto-delete
timer settings.

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

    :param message auto delete time: New auto-delete time for messages in the
chat; in seconds
    :type message_auto_delete_time: :obj:`int`

    :return: Instance of the class
    :rtype: :class:`telebot.types.MessageAutoDeleteTimerChanged`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string, dict_copy=False)
        return cls(**obj)

    def __init__(self, message_auto_delete_time, **kwargs):
        self.message_auto_delete_time = message_auto_delete_time


class MenuButton(JsonDeserializable, JsonSerializable, Dictionaryable):
    """
    This object describes the bot's menu button in a private chat. It should
be one of

    * :class:`MenuButtonCommands`
    * :class:`MenuButtonWebApp`
    * :class:`MenuButtonDefault`

    If a menu button other than MenuButtonDefault is set for a private chat,
then it is applied
    in the chat. Otherwise the default menu button is applied. By default,
the menu button opens the list of bot commands.
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        map = {
            'commands': MenuButtonCommands,
            'web_app': MenuButtonWebApp,
            'default': MenuButtonDefault
        }
        return map[obj['type']](**obj)

    def to_json(self):
        """
        :meta private:
```

```
        """
        raise NotImplementedError

    def to_dict(self):
        """
        :meta private:
        """
        raise NotImplementedError


class MenuButtonCommands(MenuButton):
    """
    Represents a menu button, which opens the bot's list of commands.

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

    :param type: Type of the button, must be commands
    :type type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.MenuButtonCommands`
    """

    def __init__(self, type):
        self.type = type

    def to_dict(self):
        return {'type': self.type}

    def to_json(self):
        return json.dumps(self.to_dict())

class MenuButtonWebApp(MenuButton):
    """
    Represents a menu button, which launches a Web App.

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

    :param type: Type of the button, must be web_app
    :type type: :obj:`str`

    :param text: Text on the button
    :type text: :obj:`str`

    :param web app: Description of the Web App that will be launched when the
user presses the button. The Web App will be
        able to send an arbitrary message on behalf of the user using the
method answerWebAppQuery.
    :type web app: :class:`telebot.types.WebAppInfo`

    :return: Instance of the class
    :rtype: :class:`telebot.types.MenuButtonWebApp`
    """

    def __init__(self, type, text, web_app):
        self.type: str = type
        self.text: str = text
        self.web_app: WebAppInfo = web_app

    def to_dict(self):
        return {'type': self.type, 'text': self.text, 'web_app':
self.web_app.to_dict()}
```

```
    def to_json(self):
        return json.dumps(self.to_dict())

class MenuButtonDefault(MenuButton):
    """
    Describes that no specific value for the menu button was set.

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

    :param type: Type of the button, must be default
    :type type: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.MenuButtonDefault`
    """
    def __init__(self, type):
        self.type: str = type

    def to_dict(self):
        return {'type': self.type}

    def to_json(self):
        return json.dumps(self.to_dict())


class ChatAdministratorRights(JsonDeserializable, JsonSerializable,
Dictionaryable):
    """
    Represents the rights of an administrator in a chat.

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

    :param is_anonymous: True, if the user's presence in the chat is hidden
    :type is_anonymous: :obj:`bool`

    :param can_manage_chat: 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
    :type can_manage_chat: :obj:`bool`

    :param can delete messages: True, if the administrator can delete
messages of other users
    :type can_delete_messages: :obj:`bool`

    :param can_manage_video_chats: True, if the administrator can manage
video chats
    :type can_manage_video_chats: :obj:`bool`

    :param can_restrict_members: True, if the administrator can restrict, ban
or unban chat members
    :type can restrict members: :obj:`bool`

    :param can_promote_members: True, if the administrator can add new
administrators with a subset of their own
        privileges or demote administrators that he has promoted, directly or
indirectly (promoted by administrators that
        were appointed by the user)
    :type can_promote_members: :obj:`bool`
```

```
    :param can_change_info: True, if the user is allowed to change the chat
title, photo and other settings
    :type can_change_info: :obj:`bool`

    :param can_invite_users: True, if the user is allowed to invite new users
to the chat
    :type can invite users: :obj:`bool`

    :param can_post_messages: Optional. True, if the administrator can post
in the channel; channels only
    :type can post messages: :obj:`bool`

    :param can_edit_messages: Optional. True, if the administrator can edit
messages of other users and can pin
        messages; channels only
    :type can edit messages: :obj:`bool`

    :param can_pin_messages: Optional. True, if the user is allowed to pin
messages; groups and supergroups only
    :type can pin messages: :obj:`bool`

    :param can_manage_topics: Optional. True, if the user is allowed to
create, rename, close, and reopen forum topics; supergroups only
    :type can manage topics: :obj:`bool`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ChatAdministratorRights`
    """

    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        return cls(**obj)

    def __init__(self, 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: bool=None, can_edit_messages: bool=None,
        can_pin_messages: bool=None, can_manage_topics: bool=None) -> None:

        self.is_anonymous: bool = is_anonymous
        self.can_manage_chat: bool = can_manage_chat
        self.can_delete_messages: bool = can_delete_messages
        self.can_manage_video_chats: bool = can_manage_video_chats
        self.can_restrict_members: bool = can_restrict_members
        self.can_promote_members: bool = can_promote_members
        self.can_change_info: bool = can_change_info
        self.can_invite_users: bool = can_invite_users
        self.can_post_messages: bool = can_post_messages
        self.can_edit_messages: bool = can_edit_messages
        self.can_pin_messages: bool = can_pin_messages
        self.can_manage_topics: bool = can_manage_topics

    def to_dict(self):
        json_dict = {
            'is_anonymous': self.is_anonymous,
            'can_manage_chat': self.can_manage_chat,
            'can_delete_messages': self.can_delete_messages,
            'can_manage_video_chats': self.can_manage_video_chats,
            'can_restrict_members': self.can_restrict_members,
            'can_promote_members': self.can_promote_members,
```

```
            'can_change_info': self.can_change_info,
            'can_invite_users': self.can_invite_users,
        }
        if self.can_post_messages is not None:
            json_dict['can_post_messages'] = self.can_post_messages
        if self.can_edit_messages is not None:
            json_dict['can_edit_messages'] = self.can_edit_messages
        if self.can_pin_messages is not None:
            json_dict['can_pin_messages'] = self.can_pin_messages
        if self.can_manage_topics is not None:
            json_dict['can_manage_topics'] = self.can_manage_topics
        return json_dict

    def to_json(self):
        return json.dumps(self.to_dict())



class InputFile:
    """
    A class to send files through Telegram Bot API.

    You need to pass a file, which should be an instance of
:class:`io.IOBase` or
    :class:`pathlib.Path`, or :obj:`str`.

    If you pass an :obj:`str` as a file, it will be opened and closed by the
class.

    :param file: A file to send.
    :type file: :class:`io.IOBase` or :class:`pathlib.Path` or :obj:`str`

    .. code-block:: python3
        :caption: Example on sending a file using this class

        from telebot.types import InputFile

        # Sending a file from disk
        bot.send_document(
            chat_id,
            InputFile('/path/to/file/file.txt')
        )


        # Sending a file from an io.IOBase object
        with open('/path/to/file/file.txt', 'rb') as f:
            bot.send_document(
                chat_id,
                InputFile(f)
            )

        # Sending a file using pathlib.Path:
        bot.send_document(
            chat_id,
            InputFile(pathlib.Path('/path/to/file/file.txt'))
        )
    """
    def __init__(self, file) -> None:
        self._file, self.file_name = self._resolve_file(file)

    def _resolve_file(self, file):
        if isinstance(file, str):
            _file = open(file, 'rb')
            return _file, os.path.basename(_file.name)
```

```
        elif isinstance(file, IOBase):
            return file, os.path.basename(file.name)
        elif isinstance(file, Path):
            _file = open(file, 'rb')
            return _file, os.path.basename(_file.name)
        else:
            raise TypeError("File must be a string or a file-like
object(pathlib.Path, io.IOBase).")


    @property
    def file(self):
        """
        File object.
        """
        return self._file


class ForumTopicCreated(JsonDeserializable):
    """
    This object represents a service message about a new forum topic created
in the chat.

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

    :param name: Name of the topic
    :type name: :obj:`str`

    :param icon_color: Color of the topic icon in RGB format
    :type icon_color: :obj:`int`

    :param icon_custom_emoji_id: Optional. Unique identifier of the custom
emoji shown as the topic icon
    :type icon_custom_emoji_id: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ForumTopicCreated`
    """
    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        return cls(**obj)

    def __init__(self, name: str, icon_color: int, icon_custom_emoji_id:
Optional[str]=None) -> None:
        self.name: str = name
        self.icon_color: int = icon_color
        self.icon_custom_emoji_id: Optional[str] = icon_custom_emoji_id


class ForumTopicClosed(JsonDeserializable):
    """
    This object represents a service message about a forum topic closed in
the chat. Currently holds no information.

    Telegram documentation:
https://core.telegram.org/bots/api#forumtopicclosed
    """
    # for future use
    @classmethod
    def de_json(cls, json_string):
        return cls()
```

```
    def __init__(self) -> None:
        pass


class ForumTopicReopened(JsonDeserializable):
    """
    This object represents a service message about a forum topic reopened in
the chat. Currently holds no information.

    Telegram documentation:
https://core.telegram.org/bots/api#forumtopicreopened
    """
    # for future use
    @classmethod
    def de_json(cls, json_string):
        return cls()

    def __init__(self) -> None:
        pass


class ForumTopic(JsonDeserializable):
    """
    This object represents a forum topic.

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

    :param message_thread_id: Unique identifier of the forum topic
    :type message_thread_id: :obj:`int`

    :param name: Name of the topic
    :type name: :obj:`str`

    :param icon_color: Color of the topic icon in RGB format
    :type icon_color: :obj:`int`

    :param icon_custom_emoji_id: Optional. Unique identifier of the custom
emoji shown as the topic icon
    :type icon_custom_emoji_id: :obj:`str`

    :return: Instance of the class
    :rtype: :class:`telebot.types.ForumTopic`
    """

    @classmethod
    def de_json(cls, json_string):
        if json_string is None: return None
        obj = cls.check_json(json_string)
        return cls(**obj)

    def __init__(self, message_thread_id: int, name: str, icon_color: int,
icon_custom_emoji_id: Optional[str]=None) -> None:
        self.message_thread_id: int = message_thread_id
        self.name: str = name
        self.icon_color: int = icon_color
        self.icon_custom_emoji_id: Optional[str] = icon_custom_emoji_id
```

Для безопасности код config.py не приводится.

Анализ результатов