telepot Documentation

Release 10.5

Nick Lee

March 02, 2017

Contents

1 telepot reference **1**
- 1.1 Basic Bot . . . 1
- 1.2 Functions . . . 5
- 1.3 DelegatorBot . . . 6
- 1.4 `telepot.delegate` . . . 7
- 1.5 `telepot.helper` . . . 9
- 1.6 `telepot.exception` . . . 17
- 1.7 `telepot.namedtuple` . . . 18
- 1.8 `telepot.routing` . . . 19

2 Installation **23**

3 Get a token **25**

4 Test the account **27**

5 Receive messages **29**

6 An easier way to receive messages **31**

7 Send a message **33**

8 Quickly `glance` a message **35**

9 Custom Keyboard and Inline Keyboard **37**

10 Message has a Flavor **39**

11 Inline Query **41**

12 Maintain Threads of Conversation **45**

13 Inline Handler per User **47**

14 Async Version (Python 3.5+) **49**
- 14.1 More Examples » . . . 50

Python Module Index **51**

CHAPTER 1

telepot reference

Telepot has two versions:

- **Traditional version works on Python 2.7 and Python 3.** It uses urllib3 to make HTTP requests, and uses threads to achieve delegation by default.
- **Async version works on Python 3.5 or above.** It is based on asyncio, uses aiohttp to make asynchronous HTTP requests, and uses asyncio tasks to achieve delegation.

This page focuses on traditional version. Async version is very similar, the most significant differences being:

- Blocking methods (mostly network operations) become coroutines, and should be called with `await`.
- Delegation is achieved by tasks, instead of threads. Thread-safety ceases to be a concern.

Traditional modules are under the package `telepot`, while async modules are under `telepot.aio`:

Traditional	Async
telepot	telepot.aio
telepot.delegate	telepot.aio.delegate
telepot.helper	telepot.aio.helper
telepot.routing	telepot.aio.routing
telepot.api	telepot.aio.api

Some modules do not have async counterparts, e.g. `telepot.namedtuple` and *`telepot.exception`*, because they are shared.

Try to combine this reading with the provided examples . One example is worth a thousand words. I hope they make things clear.

Basic Bot

The `Bot` class is mostly a wrapper around Telegram Bot API. Many methods are straight mappings to Bot API methods. Where appropriate, I only give links below. No point to duplicate all the details.

class `telepot.`**`Bot`**(*token*)

`getMe`()
See: https://core.telegram.org/bots/api#getme

`sendMessage`(*chat_id*, *text*, *parse_mode=None*, *disable_web_page_preview=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendmessage

forwardMessage(*chat_id*, *from_chat_id*, *message_id*, *disable_notification=None*)
See: https://core.telegram.org/bots/api#forwardmessage

sendPhoto(*chat_id*, *photo*, *caption=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendphoto

Parameters **photo** – a string indicating a `file_id` on server, a file-like object as obtained by `open()` or `urlopen()`, or a (filename, file-like object) tuple. If the file-like object is obtained by `urlopen()`, you most likely have to supply a filename because Telegram servers require to know the file extension. If the filename contains non-ASCII characters and you are using Python 2.7, make sure the filename is a unicode string.

sendAudio(*chat_id*, *audio*, *caption=None*, *duration=None*, *performer=None*, *title=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendaudio

Parameters **audio** – Same as `photo` in *telepot.Bot.sendPhoto()*

sendDocument(*chat_id*, *document*, *caption=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#senddocument

Parameters **document** – Same as `photo` in *telepot.Bot.sendPhoto()*

sendSticker(*chat_id*, *sticker*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendsticker

Parameters **sticker** – Same as `photo` in *telepot.Bot.sendPhoto()*

sendVideo(*chat_id*, *video*, *duration=None*, *width=None*, *height=None*, *caption=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendvideo

Parameters **video** – Same as `photo` in *telepot.Bot.sendPhoto()*

sendVoice(*chat_id*, *voice*, *caption=None*, *duration=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendvoice

Parameters **voice** – Same as `photo` in *telepot.Bot.sendPhoto()*

sendLocation(*chat_id*, *latitude*, *longitude*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendlocation

sendVenue(*chat_id*, *latitude*, *longitude*, *title*, *address*, *foursquare_id=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendvenue

sendContact(*chat_id*, *phone_number*, *first_name*, *last_name=None*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendcontact

sendGame(*chat_id*, *game_short_name*, *disable_notification=None*, *reply_to_message_id=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#sendgame

sendChatAction(*chat_id*, *action*)
See: https://core.telegram.org/bots/api#sendchataction

getUserProfilePhotos(*user_id*, *offset=None*, *limit=None*)
See: https://core.telegram.org/bots/api#getuserprofilephotos

getFile(*file_id*)
See: https://core.telegram.org/bots/api#getfile

kickChatMember(*chat_id*, *user_id*)
See: https://core.telegram.org/bots/api#kickchatmember

leaveChat(*chat_id*)
See: https://core.telegram.org/bots/api#leavechat

unbanChatMember(*chat_id*, *user_id*)
See: https://core.telegram.org/bots/api#unbanchatmember

getChat(*chat_id*)
See: https://core.telegram.org/bots/api#getchat

getChatAdministrators(*chat_id*)
See: https://core.telegram.org/bots/api#getchatadministrators

getChatMembersCount(*chat_id*)
See: https://core.telegram.org/bots/api#getchatmemberscount

getChatMember(*chat_id*, *user_id*)
See: https://core.telegram.org/bots/api#getchatmember

answerCallbackQuery(*callback_query_id*, *text=None*, *show_alert=None*, *url=None*, *cache_time=None*)
See: https://core.telegram.org/bots/api#answercallbackquery

editMessageText(*msg_identifier*, *text*, *parse_mode=None*, *disable_web_page_preview=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#editmessagetext

> **Parameters** **msg_identifier** – a 2-tuple (`chat_id`, `message_id`), a 1-tuple (`inline_message_id`), or simply `inline_message_id`. You may extract this value easily with *`telepot.message_identifier()`*

editMessageCaption(*msg_identifier*, *caption=None*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#editmessagecaption

> **Parameters** **msg_identifier** – Same as `msg_identifier` in *`telepot.Bot.editMessageText()`*

editMessageReplyMarkup(*msg_identifier*, *reply_markup=None*)
See: https://core.telegram.org/bots/api#editmessagereplymarkup

> **Parameters** **msg_identifier** – Same as `msg_identifier` in *`telepot.Bot.editMessageText()`*

answerInlineQuery(*inline_query_id*, *results*, *cache_time=None*, *is_personal=None*, *next_offset=None*, *switch_pm_text=None*, *switch_pm_parameter=None*)
See: https://core.telegram.org/bots/api#answerinlinequery

getUpdates(*offset=None*, *limit=None*, *timeout=None*, *allowed_updates=None*)
See: https://core.telegram.org/bots/api#getupdates

setWebhook(*url=None*, *certificate=None*, *max_connections=None*, *allowed_updates=None*)
See: https://core.telegram.org/bots/api#setwebhook

deleteWebhook()
See: https://core.telegram.org/bots/api#deletewebhook

getWebhookInfo()
See: https://core.telegram.org/bots/api#getwebhookinfo

setGameScore(*user_id*, *score*, *game_message_identifier*, *force=None*, *disable_edit_message=None*)
See: https://core.telegram.org/bots/api#setgamescore

Parameters **game_message_identifier** – Same as `msg_identifier` in *telepot.Bot.editMessageText()*

getGameHighScores(*user_id*, *game_message_identifier*)
See: https://core.telegram.org/bots/api#getgamehighscores

Parameters **game_message_identifier** – Same as `msg_identifier` in *telepot.Bot.editMessageText()*

download_file(*file_id*, *dest*)
Download a file to local disk.

Parameters **dest** – a path or a `file` object

message_loop(*callback=None*, *relax=0.1*, *timeout=20*, *allowed_updates=None*, *source=None*, *ordered=True*, *maxhold=3*, *run_forever=False*)
Spawn a thread to constantly `getUpdates` or pull updates from a queue. Apply `callback` to every message received. Also starts the scheduler thread for internal events.

Parameters **callback** – a function that takes one argument (the message), or a routing table. If `None`, the bot's `handle` method is used.

A *routing table* is a dictionary of `{flavor: function}`, mapping messages to appropriate handler functions according to their flavors. It allows you to define functions specifically to handle one flavor of messages. It usually looks like this: `{'chat': fn1, 'callback_query': fn2, 'inline_query': fn3, ...}`. Each handler function should take one argument (the message).

Parameters **source** – Source of updates. If `None`, `getUpdates` is used to obtain new messages from Telegram servers. If it is a synchronized queue (`Queue.Queue` in Python 2.7 or `queue.Queue` in Python 3), new messages are pulled from the queue. A web application implementing a webhook can dump updates into the queue, while the bot pulls from it. This is how telepot can be integrated with webhooks.

Acceptable contents in queue:

- `str`, `unicode` (Python 2.7), or `bytes` (Python 3, decoded using UTF-8) representing a JSON-serialized Update object.
- a `dict` representing an Update object.

When `source` is `None`, these parameters are meaningful:

Parameters

- **relax** (*float*) – seconds between each `getUpdates`
- **timeout** (*int*) – `timeout` parameter supplied to *telepot.Bot.getUpdates()*, controlling how long to poll.
- **allowed_updates** (*array of string*) – `allowed_updates` parameter supplied to *telepot.Bot.getUpdates()*, controlling which types of updates to receive.

When `source` is a queue, these parameters are meaningful:

Parameters

- **ordered** (*bool*) – If `True`, ensure in-order delivery of messages to `callback` (i.e. updates with a smaller `update_id` always come before those with a larger `update_id`). If `False`, no re-ordering is done. `callback` is applied to messages as soon as they are pulled from queue.

- **maxhold** (*float*) – Applied only when `ordered` is `True`. The maximum number of seconds an update is held waiting for a not-yet-arrived smaller `update_id`. When this number of seconds is up, the update is delivered to `callback` even if some smaller `update_id`s have not yet arrived. If those smaller `update_id`s arrive at some later time, they are discarded.

Finally, there is this parameter, meaningful always:

Parameters **run_forever** (*bool or str*) – If `True` or any non-empty string, append an infinite loop at the end of this method, so it never returns. Useful as the very last line in a program. A non-empty string will also be printed, useful as an indication that the program is listening.

Functions

`telepot.`**`flavor`**(*msg*)

Return flavor of message or event.

A message's flavor may be one of these:

- `chat`
- `callback_query`
- `inline_query`
- `chosen_inline_result`

An event's flavor is determined by the single top-level key.

`telepot.`**`glance`**(*msg*, *flavor='chat'*, *long=False*)

Extract "headline" info about a message. Use parameter `long` to control whether a short or long tuple is returned.

When `flavor` is `chat` (`msg` being a Message object):

- short: (content_type, `msg['chat']['type']`, `msg['chat']['id']`)
- long: (content_type, `msg['chat']['type']`, `msg['chat']['id']`, `msg['date']`, `msg['message_id']`)

content_type can be: `text`, `audio`, `document`, `game`, `photo`, `sticker`, `video`, `voice`, `contact`, `location`, `venue`, `new_chat_member`, `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`.

When `flavor` is `callback_query` (`msg` being a CallbackQuery object):

- regardless: (`msg['id']`, `msg['from']['id']`, `msg['data']`)

When `flavor` is `inline_query` (`msg` being a InlineQuery object):

- short: (`msg['id']`, `msg['from']['id']`, `msg['query']`)
- long: (`msg['id']`, `msg['from']['id']`, `msg['query']`, `msg['offset']`)

When `flavor` is `chosen_inline_result` (`msg` being a ChosenInlineResult object):

- regardless: (`msg['result_id']`, `msg['from']['id']`, `msg['query']`)

`telepot.`**`flance`**(*msg*, *long=False*)
A combination of *`telepot.flavor()`* and *`telepot.glance()`*, return a 2-tuple (flavor, headline_info), where *headline_info* is whatever extracted by *`telepot.glance()`* depending on the message flavor and the `long` parameter.

`telepot.`**`peel`**(*event*)
Remove an event's top-level skin (where its flavor is determined), and return the core content.

`telepot.`**`fleece`**(*event*)
A combination of *`telepot.flavor()`* and *`telepot.peel()`*, return a 2-tuple (flavor, content) of an event.

`telepot.`**`is_event`**(*msg*)
Return whether the message looks like an event. That is, whether it has a flavor that starts with an underscore.

`telepot.`**`message_identifier`**(*msg*)
Extract an identifier for message editing. Useful with *`telepot.Bot.editMessageText()`* and similar methods. Returned value is guaranteed to be a tuple.

`msg` is expected to be `chat` or `choson_inline_result`.

`telepot.`**`origin_identifier`**(*msg*)
Extract the message identifier of a callback query's origin. Returned value is guaranteed to be a tuple.

`msg` is expected to be `callback_query`.

DelegatorBot

class `telepot.`**`DelegatorBot`**(*token*, *delegation_patterns*)

Parameters **`delegation_patterns`** – a list of (seeder, delegator) tuples.

A *seeder* is a function that:

- takes one argument - a message
- **returns a *seed*. Depending on the nature of the seed, behavior is as follows:**
  - if the seed is a hashable (e.g. number, string, tuple), it looks for a *delegate* associated with the seed. (Think of a dictionary of `{seed: delegate}`)
    - if such a delegate exists and is alive, it is assumed that the message will be picked up by the delegate. Nothing more is done.
    - if no delegate exists or that delegate is no longer alive, a new delegate is obtained by calling the delegator function. The new delegate is associated with the seed.
    - **In essence, when the seed is a hashable, only one delegate is running for a given seed.**
  - if the seed is a non-hashable, (e.g. list), a new delegate is always obtained by calling the delegator function. No seed-delegate association occurs.
  - if the seed is `None`, nothing is done.

A *delegator* is a function that:

- takes one argument - a (bot, message, seed) tuple. This is called a *seed tuple*.
- **returns a *delegate*, which can be one of the following:**
  - an object that has methods `start()` and `is_alive()`. Therefore, a `threading.Thread` object is a natural delegate. Once returned, the object's `start()` method is called.

- a function. Once returned, it is wrapped in a `Thread(target=function)` and started.
- a (function, args, kwargs) tuple. Once returned, it is wrapped in a `Thread(target=function, args=args, kwargs=kwargs)` and started.

The above logic is implemented in the `handle` method. You only have to call *`Bot.message_loop()`* with no `callback` argument, the above logic will be executed for every message received.

In the list of delegation patterns, all seeder functions are evaluated in order. One message may start multiple delegates.

The module *`telepot.delegate`* has a bunch of seeder factories and delegator factories, which greatly ease the use of DelegatorBot. The module `telepot.helper` also has a number of `*Handler` classes which provide a connection-like interface to deal with individual chats or users.

In the rest of discussions, *seed tuple* means a (bot, message, seed) tuple, referring to the single argument taken by delegator functions.

telepot.delegate

`telepot.delegate.`**`per_chat_id`**(*types='all'*)

Parameters **`types`** – `all` or a list of chat types (`private`, `group`, `channel`)

Returns a seeder function that returns the chat id only if the chat type is in `types`.

`telepot.delegate.`**`per_chat_id_in`**(*s*, *types='all'*)

Parameters

- **`s`** – a list or set of chat id
- **`types`** – `all` or a list of chat types (`private`, `group`, `channel`)

Returns a seeder function that returns the chat id only if the chat id is in `s` and chat type is in `types`.

`telepot.delegate.`**`per_chat_id_except`**(*s*, *types='all'*)

Parameters

- **`s`** – a list or set of chat id
- **`types`** – `all` or a list of chat types (`private`, `group`, `channel`)

Returns a seeder function that returns the chat id only if the chat id is *not* in `s` and chat type is in `types`.

`telepot.delegate.`**`per_from_id`**(*flavors=['chat', 'inline_query', 'chosen_inline_result']*)

Parameters **`flavors`** – `all` or a list of flavors

Returns a seeder function that returns the from id only if the message flavor is in `flavors`.

`telepot.delegate.`**`per_from_id_in`**(*s, flavors=['chat', 'inline_query', 'chosen_inline_result']*)

Parameters

- **`s`** – a list or set of from id
- **`flavors`** – `all` or a list of flavors

Returns a seeder function that returns the from id only if the from id is in `s` and message flavor is in `flavors`.

`telepot.delegate.`**`per_from_id_except`**(*s, flavors=['chat', 'inline_query', 'chosen_inline_result']*)

Parameters

- **s** – a list or set of from id
- **flavors** – `all` or a list of flavors

Returns a seeder function that returns the from id only if the from id is *not* in `s` and message flavor is in `flavors`.

`telepot.delegate.`**`per_inline_from_id`**`()`

Returns a seeder function that returns the from id only if the message flavor is `inline_query` or `chosen_inline_result`

`telepot.delegate.`**`per_inline_from_id_in`**`(`*s*`)`

Parameters **s** – a list or set of from id

Returns a seeder function that returns the from id only if the message flavor is `inline_query` or `chosen_inline_result` and the from id is in `s`.

`telepot.delegate.`**`per_inline_from_id_except`**`(`*s*`)`

Parameters **s** – a list or set of from id

Returns a seeder function that returns the from id only if the message flavor is `inline_query` or `chosen_inline_result` and the from id is *not* in `s`.

`telepot.delegate.`**`per_application`**`()`

Returns a seeder function that always returns 1, ensuring at most one delegate is ever spawned for the entire application.

`telepot.delegate.`**`per_message`**`(`*flavors='all'*`)`

Parameters **flavors** – `all` or a list of flavors

Returns a seeder function that returns a non-hashable only if the message flavor is in `flavors`.

`telepot.delegate.`**`per_event_source_id`**`(`*event_space*`)`

Returns a seeder function that returns an event's source id only if that event's source space equals to `event_space`.

`telepot.delegate.`**`per_callback_query_chat_id`**`(`*types='all'*`)`

Parameters **types** – `all` or a list of chat types (`private`, `group`, `channel`)

Returns a seeder function that returns a callback query's originating chat id if the chat type is in `types`.

`telepot.delegate.`**`per_callback_query_origin`**`(`*origins='all'*`)`

Parameters **origins** – `all` or a list of origin types (`chat`, `inline`)

Returns a seeder function that returns a callback query's origin identifier if that origin type is in `origins`. The origin identifier is guaranteed to be a tuple.

`telepot.delegate.`**`call`**`(`*func*, **args*, ***kwargs*`)`

Returns a delegator function that returns a tuple (`func`, (seed tuple,)+ `args`, `kwargs`). That is, seed tuple is inserted before supplied positional arguments. By default, a thread wrapping `func` and all those arguments is spawned.

`telepot.delegate.`**`create_run`**`(`*cls*, **args*, ***kwargs*`)`

Returns a delegator function that calls the `cls` constructor whose arguments being a seed tuple followed by supplied `*args` and `**kwargs`, then returns the object's `run` method. By default, a thread wrapping that `run` method is spawned.

`telepot.delegate.`**`create_open`**(*cls*, **args*, ***kwargs*)

Returns a delegator function that calls the `cls` constructor whose arguments being a seed tuple followed by supplied `*args` and `**kwargs`, then returns a looping function that uses the object's `listener` to wait for messages and invokes instance method `open`, `on_message`, and `on_close` accordingly. By default, a thread wrapping that looping function is spawned.

`telepot.delegate.`**`until`**(*condition*, *fns*)

Try a list of seeder functions until a condition is met.

Parameters

- **`condition`** – a function that takes one argument - a seed - and returns `True` or `False`
- **`fns`** – a list of seeder functions

Returns a "composite" seeder function that calls each supplied function in turn, and returns the first seed where the condition is met. If the condition is never met, it returns `None`.

`telepot.delegate.`**`chain`**(**fns*)

Returns a "composite" seeder function that calls each supplied function in turn, and returns the first seed that is not `None`.

`telepot.delegate.`**`pair`**(*seeders*, *delegator_factory*, **args*, ***kwargs*)

The basic pair producer.

Returns a (seeder, delegator_factory(*args, **kwargs)) tuple.

Parameters **`seeders`** – If it is a seeder function or a list of one seeder function, it is returned as the final seeder. If it is a list of more than one seeder function, they are chained together before returned as the final seeder.

`telepot.delegate.`**`pave_event_space`**(*fn=<function pair>*)

Returns a pair producer that ensures the seeder and delegator share the same event space.

`telepot.delegate.`**`include_callback_query_chat_id`**(*fn=<function pair>*, *types='all'*)

Returns a pair producer that enables static callback query capturing across seeder and delegator.

Parameters **`types`** – `all` or a list of chat types (`private`, `group`, `channel`)

`telepot.delegate.`**`intercept_callback_query_origin`**(*fn=<function pair>*, *origins='all'*)

Returns a pair producer that enables dynamic callback query origin mapping across seeder and delegator.

Parameters **`origins`** – `all` or a list of origin types (`chat`, `inline`). Origin mapping is only enabled for specified origin types.

telepot.helper

Handlers

class `telepot.helper.`**`Monitor`**(*seed_tuple*, *capture*, ***kwargs*)

Bases: *telepot.helper.ListenerContext*, *telepot.helper.DefaultRouterMixin*

A delegate that never times-out, probably doing some kind of background monitoring in the application. Most naturally paired with *per_application()*.

Parameters **capture** – a list of patterns for *Listener* to capture

class telepot.helper.**ChatHandler**(*seed_tuple*, *include_callback_query=False*, ***kwargs*)
Bases: *telepot.helper.ChatContext*, *telepot.helper.DefaultRouterMixin*, *telepot.helper.StandardEventMixin*, *telepot.helper.IdleTerminateMixin*

A delegate to handle a chat.

class telepot.helper.**UserHandler**(*seed_tuple, include_callback_query=False, flavors=['chat', 'inline_query', 'chosen_inline_result'], **kwargs*)
Bases: *telepot.helper.UserContext*, *telepot.helper.DefaultRouterMixin*, *telepot.helper.StandardEventMixin*, *telepot.helper.IdleTerminateMixin*

A delegate to handle a user's actions.

Parameters **flavors** – A list of flavors to capture. `all` covers all flavors.

class telepot.helper.**InlineUserHandler**(*seed_tuple*, ***kwargs*)
Bases: *telepot.helper.UserHandler*

A delegate to handle a user's inline-related actions.

class telepot.helper.**CallbackQueryOriginHandler**(*seed_tuple*, ***kwargs*)
Bases: *telepot.helper.CallbackQueryOriginContext*, *telepot.helper.DefaultRouterMixin*, *telepot.helper.StandardEventMixin*, *telepot.helper.IdleTerminateMixin*

A delegate to handle callback query from one origin.

Contexts

class telepot.helper.**ListenerContext**(*bot*, *context_id*, **args*, ***kwargs*)

bot
The underlying *Bot* or an augmented version thereof

id

listener
See *Listener*

class telepot.helper.**ChatContext**(*bot*, *context_id*, **args*, ***kwargs*)
Bases: *telepot.helper.ListenerContext*

chat_id

sender
A *Sender* for this chat

administrator
An *Administrator* for this chat

class telepot.helper.**UserContext**(*bot*, *context_id*, **args*, ***kwargs*)
Bases: *telepot.helper.ListenerContext*

user_id

sender
A *Sender* for this user

class telepot.helper.**CallbackQueryOriginContext**(*bot*, *context_id*, **args*, ***kwargs*)
Bases: *telepot.helper.ListenerContext*

origin
Mesasge identifier of callback query's origin

editor
An *Editor* to the originating message

class telepot.helper.**Sender**(*bot*, *chat_id*)
When you are dealing with a particular chat, it is tedious to have to supply the same `chat_id` every time to send a message, or to send anything.

This object is a proxy to a bot's `send*` and `forwardMessage` methods, automatically fills in a fixed chat id for you. Available methods have identical signatures as those of the underlying bot, **except there is no need to supply the aforementioned** `chat_id`:

- *Bot.sendMessage()*
- *Bot.forwardMessage()*
- *Bot.sendPhoto()*
- *Bot.sendAudio()*
- *Bot.sendDocument()*
- *Bot.sendSticker()*
- *Bot.sendVideo()*
- *Bot.sendVoice()*
- *Bot.sendLocation()*
- *Bot.sendVenue()*
- *Bot.sendContact()*
- *Bot.sendGame()*
- *Bot.sendChatAction()*

class telepot.helper.**Administrator**(*bot*, *chat_id*)
When you are dealing with a particular chat, it is tedious to have to supply the same `chat_id` every time to get a chat's info or to perform administrative tasks.

This object is a proxy to a bot's chat administration methods, automatically fills in a fixed chat id for you. Available methods have identical signatures as those of the underlying bot, **except there is no need to supply the aforementioned** `chat_id`:

- *Bot.kickChatMember()*
- *Bot.leaveChat()*
- *Bot.unbanChatMember()*
- *Bot.getChat()*
- *Bot.getChatAdministrators()*
- *Bot.getChatMembersCount()*
- *Bot.getChatMember()*

class telepot.helper.**Editor**(*bot*, *msg_identifier*)
If you want to edit a message over and over, it is tedious to have to supply the same `msg_identifier` every time.

This object is a proxy to a bot's message-editing methods, automatically fills in a fixed message identifier for you. Available methods have identical signatures as those of the underlying bot, **except there is no need to supply the aforementioned** `msg_identifier`:

- *Bot.editMessageText()*
- *Bot.editMessageCaption()*
- *Bot.editMessageReplyMarkup()*

A message's identifier can be easily extracted with *telepot.message_identifier()*.

Parameters **msg_identifier** – a message identifier as mentioned above, or a message (whose identifier will be automatically extracted).

class telepot.helper.**Listener**(*mic*, *q*)

capture(*pattern*)
Add a pattern to capture.

Parameters **pattern** – a list of templates.

A template may be a function that:

- takes one argument - a message
- returns `True` to indicate a match

A template may also be a dictionary whose:

- **keys** are used to *select* parts of message. Can be strings or regular expressions (as obtained by `re.compile()`)
- **values** are used to match against the selected parts. Can be typical data or a function.

All templates must produce a match for a message to be considered a match.

wait()
Block until a matched message appears.

Mixins

class telepot.helper.**Router**(*key_function*, *routing_table*)
Map a message to a handler function, using a **key function** and a **routing table** (dictionary).

A *key function* digests a message down to a value. This value is treated as a key to the *routing table* to look up a corresponding handler function.

Parameters

- **key_function** – A function that takes one argument (the message) and returns one of the following:
  - a key to the routing table
  - a 1-tuple (key,)
  - a 2-tuple (key, (positional, arguments, ...))

– a 3-tuple (key, (positional, arguments, ...), {keyword: arguments, ...})

Extra arguments, if returned, will be applied to the handler function after using the key to look up the routing table.

- **routing_table** – A dictionary of `{key: handler}`. A `None` key acts as a default catch-all. If the key being looked up does not exist in the routing table, the `None` key and its corresponding handler is used.

map(*msg*)
Apply key function to `msg` to obtain a key. Return the routing table entry.

route(*msg*, **aa*, ***kw*)
Apply key function to `msg` to obtain a key, look up routing table to obtain a handler function, then call the handler function with positional and keyword arguments, if any is returned by the key function.

`*aa` and `**kw` are dummy placeholders for easy chaining. Regardless of any number of arguments returned by the key function, multi-level routing may be achieved like this:

```
top_router.routing_table['key1'] = sub_router1.route
top_router.routing_table['key2'] = sub_router2.route
```

class `telepot.helper.`**DefaultRouterMixin**(**args*, ***kwargs*)
Install a default *Router* and the instance method `on_message()`.

router

on_message(*msg*)
Call *Router.route()* to handle the message.

class `telepot.helper.`**StandardEventScheduler**(*scheduler*, *event_space*, *source_id*)
A proxy to the underlying *Bot*'s scheduler, this object implements the *standard event format*. A standard event looks like this:

```
{'_flavor': {
    'source': {
        'space': event_space, 'id': source_id}
    'custom_key1': custom_value1,
    'custom_key2': custom_value2,
     ... }}
```

- There is a single top-level key indicating the flavor, starting with an _underscore.
- On the second level, there is a `source` key indicating the event source.
- An event source consists of an *event space* and a *source id*.
- An event space is shared by all delegates in a group. Source id simply refers to a delegate's id. They combine to ensure a delegate is always able to capture its own events, while its own events would not be mistakenly captured by others.

Events scheduled through this object always have the second-level `source` key fixed, while the flavor and other data may be customized.

event_space

configure(*listener*)
Configure a *Listener* to capture events with this object's event space and source id.

make_event_data(*flavor*, *data*)
Marshall `flavor` and `data` into a standard event.

event_at(*when*, *data_tuple*)
Schedule an event to be emitted at a certain time.

Parameters

- **when** – an absolute timestamp
- **data_tuple** – a 2-tuple (flavor, data)

Returns an event object, useful for cancelling.

event_later(*delay*, *data_tuple*)
Schedule an event to be emitted after a delay.

Parameters

- **delay** – number of seconds
- **data_tuple** – a 2-tuple (flavor, data)

Returns an event object, useful for cancelling.

event_now(*data_tuple*)
Schedule an event to be emitted now.

Parameters **data_tuple** – a 2-tuple (flavor, data)

Returns an event object, useful for cancelling.

cancel(*event*)
Cancel an event.

class telepot.helper.**StandardEventMixin**(*event_space*, **args*, ***kwargs*)
Install a *StandardEventScheduler*.

scheduler

class telepot.helper.**IdleEventCoordinator**(*scheduler*, *timeout*)

refresh()
Refresh timeout timer

augment_on_message(*handler*)

Returns a function wrapping `handler` to refresh timer for every non-event message

augment_on_close(*handler*)

Returns a function wrapping `handler` to cancel timeout event

class telepot.helper.**IdleTerminateMixin**(*timeout*, **args*, ***kwargs*)
Install an *IdleEventCoordinator* to manage idle timeout. Also define instance method `on__idle()` to handle idle timeout events.

idle_event_coordinator

on__idle(*event*)
Raise an *IdleTerminate* to close the delegate.

class telepot.helper.**CallbackQueryCoordinator**(*id*, *origin_set*, *enable_chat*, *enable_inline*)

Parameters

- **origin_set** – Callback query whose origin belongs to this set will be captured
- **enable_chat** –

- `False`: Do not intercept *chat-originated* callback query
- `True`: Do intercept
- Notifier function: Do intercept and call the notifier function on adding or removing an origin

- **`enable_inline`** – Same meaning as `enable_chat`, but apply to *inline-originated* callback query

Notifier functions should have the signature `notifier(origin, id, adding)`:

•On adding an origin, `notifier(origin, my_id, True)` will be called.

•On removing an origin, `notifier(origin, my_id, False)` will be called.

`configure`(*listener*)
Configure a *Listener* to capture callback query

`capture_origin`(*msg_identifier*, *notify=True*)

`uncapture_origin`(*msg_identifier*, *notify=True*)

`augment_send`(*send_func*)

Parameters **`send_func`** – functions that send messages, such as `Bot.send*()`

Returns a function that wraps around `send_func` and examines whether the sent message contains an inline keyboard with callback data. If so, future callback query originating from the sent message will be captured.

`augment_edit`(*edit_func*)

Parameters **`edit_func`** – functions that edit messages, such as `Bot.edit*()`

Returns a function that wraps around `edit_func` and examines whether the edited message contains an inline keyboard with callback data. If so, future callback query originating from the edited message will be captured. If not, such capturing will be stopped.

`augment_on_message`(*handler*)

Parameters **`handler`** – an `on_message()` handler function

Returns a function that wraps around `handler` and examines whether the incoming message is a chosen inline result with an `inline_message_id` field. If so, future callback query originating from this chosen inline result will be captured.

`augment_bot`(*bot*)

Returns

a proxy to `bot` with these modifications:

- all `send*` methods augmented by `augment_send()`
- all `edit*` methods augmented by `augment_edit()`
- all other public methods, including properties, copied unchanged

class `telepot.helper.`**`InterceptCallbackQueryMixin`**(*intercept_callback_query*, **args*, ***kwargs*)
Install a *CallbackQueryCoordinator* to capture callback query dynamically.

Using this mixin has one consequence. The `self.bot()` property no longer returns the original *Bot* object. Instead, it returns an augmented version of the *Bot* (augmented by *CallbackQueryCoordinator*). The original *Bot* can be accessed with `self.__bot` (double underscore).

Parameters **intercept_callback_query** – a 2-tuple (enable_chat, enable_inline) to pass to *CallbackQueryCoordinator*

callback_query_coordinator

class telepot.helper.**Answerer**(*bot*)
When processing inline queries, ensure **at most one active thread** per user id.

answer(*outerself*, *inline_query*, *compute_fn*, **compute_args*, ***compute_kwargs*)
Spawns a thread that calls `compute fn` (along with additional arguments `*compute_args` and `**compute_kwargs`), then applies the returned value to *Bot.answerInlineQuery()* to answer the inline query. If a preceding thread is already working for a user, that thread is cancelled, thus ensuring at most one active thread per user id.

Parameters

- **inline_query** – The inline query to be processed. The originating user is inferred from `msg['from']['id']`.
- **compute_fn** – A **thread-safe** function whose returned value is given to *Bot.answerInlineQuery()* to send. May return:
  - a *list* of InlineQueryResult
  - a *tuple* whose first element is a list of InlineQueryResult, followed by positional arguments to be supplied to *Bot.answerInlineQuery()*
  - a *dictionary* representing keyword arguments to be supplied to *Bot.answerInlineQuery()*
- ***compute_args** – positional arguments to `compute_fn`
- ****compute_kwargs** – keyword arguments to `compute_fn`

class telepot.helper.**AnswererMixin**(**args*, ***kwargs*)
Install an *Answerer* to handle inline query.

answerer

Utilities

class telepot.helper.**SafeDict**(**args*, ***kwargs*)
A subclass of `dict`, thread-safety added:

```
d = SafeDict()  # Thread-safe operations include:
d['a'] = 3      # key assignment
d['a']          # key retrieval
del d['a']      # key deletion
```

telepot.helper.**openable**(*cls*)
A class decorator to fill in certain methods and properties to ensure a class can be used by *create_open()*.

These instance methods and property will be added, if not defined by the class:

- `open(self, initial_msg, seed)`
- `on_message(self, msg)`
- `on_close(self, ex)`
- `close(self, ex=None)`
- property `listener`

telepot.exception

exception telepot.exception.**TelepotException**
Base class of following exceptions.

exception telepot.exception.**BadFlavor**(*offender*)

offender

exception telepot.exception.**BadHTTPResponse**(*status*, *text*, *response*)
All requests to Bot API should result in a JSON response. If non-JSON, this exception is raised. While it is hard to pinpoint exactly when this might happen, the following situations have been observed to give rise to it:

- an unreasonable token, e.g. `abc`, `123`, anything that does not even remotely resemble a correct token.
- a bad gateway, e.g. when Telegram servers are down.

status

text

response

exception telepot.exception.**EventNotFound**(*event*)

event

exception telepot.exception.**WaitTooLong**(*seconds*)

seconds

exception telepot.exception.**IdleTerminate**(*seconds*)

exception telepot.exception.**StopListening**

exception telepot.exception.**TelegramError**(*description*, *error_code*, *json*)
To indicate erroneous situations, Telegram returns a JSON object containing an *error code* and a *description*. This will cause a `TelegramError` to be raised. Before raising a generic `TelegramError`, telepot looks for a more specific subclass that "matches" the error. If such a class exists, an exception of that specific subclass is raised. This allows you to either catch specific errors or to cast a wide net (by a catch-all `TelegramError`). This also allows you to incorporate custom `TelegramError` easily.

Subclasses must define a class variable `DESCRIPTION_PATTERNS` which is a list of regular expressions. If an error's *description* matches any of the regular expressions, an exception of that subclass is raised.

description

error_code

json

exception telepot.exception.**UnauthorizedError**(*description*, *error_code*, *json*)

DESCRIPTION_PATTERNS = ['unauthorized']

exception telepot.exception.**BotWasKickedError**(*description*, *error_code*, *json*)

DESCRIPTION_PATTERNS = ['bot.*kicked']

exception telepot.exception.**BotWasBlockedError**(*description*, *error_code*, *json*)

DESCRIPTION_PATTERNS = ['bot.*blocked']

exception telepot.exception.**TooManyRequestsError**(*description*, *error_code*, *json*)

DESCRIPTION_PATTERNS = ['too *many *requests']

exception telepot.exception.**MigratedToSupergroupChatError**(*description*, *error_code*, *json*)

DESCRIPTION_PATTERNS = ['migrated.*supergroup *chat']

telepot.namedtuple

Telepot's custom is to represent Bot API object as *dictionary*. On the other hand, the module `telepot.namedtuple` also provide namedtuple classes mirroring those objects. The reasons are twofold:

1. Under some situations, you may want an object with a complete set of fields, including those whose values are `None`. A dictionary translated from Bot API's response would have those `None` fields absent. By converting such a dictionary to a namedtuple, all fields are guaranteed to be present, even if their values are `None`. This usage is for **incoming** objects received from Telegram servers.
2. Namedtuple allows easier construction of objects like ReplyKeyboardMarkup, InlineKeyboardMarkup, and various InlineQueryResult, etc. This usage is for **outgoing** objects sent to Telegram servers.

Incoming objects include:

- User
- Chat
- Message
- MessageEntity
- PhotoSize
- Audio
- Document
- Sticker
- Video
- Voice
- Contact
- Location
- Venue
- UserProfilePhotos
- File
- ChatMember
- CallbackQuery
- InlineQuery

- ChosenInlineResult

Outgoing objects include:

- ReplyKeyboardMarkup
- KeyboardButton
- ReplyKeyboardRemove
- InlineKeyboardMarkup
- InlineKeyboardButton
- ForceReply
- Various types of InlineQueryResult
- Various types of InputMessageContent

telepot.routing

This module has a bunch of key function factories and routing table factories to facilitate the use of *Router*.

Things to remember:

1. A key function takes one argument - the message, and returns a key, optionally followed by positional arguments and keyword arguments.
2. A routing table is just a dictionary. After obtaining one from a factory function, you can customize it to your liking.

telepot.routing.**by_content_type**()

Returns A key function that returns a 2-tuple (content_type, (msg[content_type],)). In plain English, it returns the message's *content type* as the key, and the corresponding content as a positional argument to the handler function.

telepot.routing.**by_command**(*extractor*, *prefix=('/',)*, *separator=' '*, *pass_args=False*)

Parameters

- **extractor** – a function that takes one argument (the message) and returns a portion of message to be interpreted. To extract the text of a chat message, use `lambda msg: msg['text']`.
- **prefix** – a list of special characters expected to indicate the head of a command.
- **separator** – a command may be followed by arguments separated by `separator`.
- **pass_args** (*bool*) – If `True`, arguments following a command will be passed to the handler function.

Returns a key function that interprets a specific part of a message and returns the embedded command, optionally followed by arguments. If the text is not preceded by any of the specified `prefix`, it returns a 1-tuple `(None,)` as the key. This is to distinguish with the special `None` key in routing table.

telepot.routing.**by_chat_command**(*prefix=('/',)*, *separator=' '*, *pass_args=False*)

Parameters

- **prefix** – a list of special characters expected to indicate the head of a command.
- **separator** – a command may be followed by arguments separated by `separator`.

- **pass_args** (*bool*) – If `True`, arguments following a command will be passed to the handler function.

Returns a key function that interprets a chat message's text and returns the embedded command, optionally followed by arguments. If the text is not preceded by any of the specified `prefix`, it returns a 1-tuple `(None,)` as the key. This is to distinguish with the special `None` key in routing table.

`telepot.routing.`**`by_text`**`()`

Returns a key function that returns a message's `text` field.

`telepot.routing.`**`by_data`**`()`

Returns a key function that returns a message's `data` field.

`telepot.routing.`**`by_regex`**`(`*extractor*, *regex*, *key=1*`)`

Parameters

- **extractor** – a function that takes one argument (the message) and returns a portion of message to be interpreted. To extract the text of a chat message, use `lambda msg: msg['text']`.
- **regex** (*str or regex object*) – the pattern to look for
- **key** – the part of match object to be used as key

Returns a key function that returns `match.group(key)` as key (where `match` is the match object) and the match object as a positional argument. If no match is found, it returns a 1-tuple `(None,)` as the key. This is to distinguish with the special `None` key in routing table.

`telepot.routing.`**`process_key`**`(`*processor*, *fn*`)`

Parameters

- **processor** – a function to process the key returned by the supplied key function
- **fn** – a key function

Returns a function that wraps around the supplied key function to further process the key before returning.

`telepot.routing.`**`lower_key`**`(`*fn*`)`

Parameters **fn** – a key function

Returns a function that wraps around the supplied key function to ensure the returned key is in lowercase.

`telepot.routing.`**`upper_key`**`(`*fn*`)`

Parameters **fn** – a key function

Returns a function that wraps around the supplied key function to ensure the returned key is in uppercase.

`telepot.routing.`**`make_routing_table`**`(`*obj*, *keys*, *prefix='on_'*`)`

Returns a dictionary roughly equivalent to `{'key1': obj.on_key1, 'key2': obj.on_key2, ...}`, but `obj` does not have to define all methods. It may define the needed ones only.

Parameters

- **obj** – the object

- **keys** – a list of keys
- **prefix** – a string to be prepended to keys to make method names

telepot.routing.**make_content_type_routing_table**(*obj*, *prefix='on_'*)

Returns a dictionary covering all available content types, roughly equivalent to `{'text': obj.on_text, 'photo': obj.on_photo, ...}`, but `obj` does not have to define all methods. It may define the needed ones only.

Parameters

- **obj** – the object
- **prefix** – a string to be prepended to content types to make method names

Telepot helps you build applications for Telegram Bot API. It works on Python 2.7 and Python 3. It also has an *async version* based on asyncio and Python 3.5+.

For a time, I tried to list the features here like many projects do. Eventually, I gave up.

For common and straight-forward features, I find them too trivial to worth listing. For more unique and novel features, I cannot find standard terms to describe them. The best way to experience telepot is by reading this page and going through the examples. Let's go.

CHAPTER 2

Installation

pip:

```
$ pip install telepot
$ pip install telepot --upgrade  # UPGRADE
```

easy_install:

```
$ easy_install telepot
$ easy_install --upgrade telepot  # UPGRADE
```

CHAPTER 3

Get a token

To use the Telegram Bot API, you first have to get a bot account by chatting with BotFather.

BotFather will give you a **token**, something like `123456789:ABCdefGhIJKlmNoPQRsTUVwxyZ`. With the token in hand, you can start using telepot to access the bot account.

CHAPTER 4

Test the account

```
>>> import telepot
>>> bot = telepot.Bot('***** PUT YOUR TOKEN HERE *****')
>>> bot.getMe()
{'first_name': 'Your Bot', 'username': 'YourBot', 'id': 123456789}
```

CHAPTER 5

Receive messages

Bots cannot initiate conversations with users. You have to send it a message first. Get the message by calling *Bot.getUpdates()*:

```
>>> from pprint import pprint
>>> response = bot.getUpdates()
>>> pprint(response)
[{'message': {'chat': {'first_name': 'Nick',
                       'id': 999999999,
                       'type': 'private'},
              'date': 1465283242,
              'from': {'first_name': 'Nick', 'id': 999999999},
              'message_id': 10772,
              'text': 'Hello'},
  'update_id': 100000000}]
```

`999999999` is obviously a fake id. `Nick` is my real name, though.

The `chat` field represents the conversation. Its `type` can be `private`, `group`, or `channel` (whose meanings should be obvious, I hope). Above, `Nick` just sent a `private` message to the bot.

According to Bot API, the method getUpdates returns an array of Update objects. As you can see, an Update object is nothing more than a Python dictionary. In telepot, **Bot API objects are represented as dictionary.**

Note the `update_id`. It is an ever-increasing number. Next time you should use `getUpdates(offset=100000001)` to avoid getting the same old messages over and over. Giving an `offset` essentially acknowledges to the server that you have received all `update_id`s lower than `offset`:

```
>>> bot.getUpdates(offset=100000001)
[]
```

CHAPTER 6

An easier way to receive messages

It is troublesome to keep checking messages while managing `offset`. Let telepot take care of the mundane stuff and notify you whenever new messages arrive:

```
>>> def handle(msg):
...     pprint(msg)
...
>>> bot.message_loop(handle)
```

After setting up this callback, send it a few messages. Sit back and monitor the messages arriving.

CHAPTER 7

Send a message

Sooner or later, your bot will want to send *you* messages. You should have discovered your own user id from above interactions. I will keeping using my fake id of `999999999`. Remember to substitute your own (real) id:

```
>>> bot.sendMessage(999999999, 'Hey!')
```

CHAPTER 8

Quickly `glance` a message

When processing a message, a few pieces of information are so central that you almost always have to extract them. Use *`telepot.glance()`* to extract "headline info". Try this skeleton, a bot which echoes what you said:

```
import sys
import time
import telepot

def handle(msg):
    content_type, chat_type, chat_id = telepot.glance(msg)
    print(content_type, chat_type, chat_id)

    if content_type == 'text':
        bot.sendMessage(chat_id, msg['text'])

TOKEN = sys.argv[1]  # get token from command-line

bot = telepot.Bot(TOKEN)
bot.message_loop(handle)
print ('Listening ...')

# Keep the program running.
while 1:
    time.sleep(10)
```

It is a good habit to always check `content_type` before further processing. Do not assume every message is a `text`.

CHAPTER 9

Custom Keyboard and Inline Keyboard

Besides sending messages back and forth, Bot API allows richer interactions with custom keyboard and inline keyboard. Both can be specified with the parameter `reply_markup` in *`Bot.sendMessage()`*. The module `telepot.namedtuple` provides namedtuple classes for easier construction of these keyboards.

Pressing a button on a *custom* keyboard results in a Message object sent to the bot, which is no different from a regular chat message composed by typing.

Pressing a button on an *inline* keyboard results in a CallbackQuery object sent to the bot, which we have to distinguish from a Message object.

Here comes the concept of **flavor**.

CHAPTER 10

Message has a Flavor

Regardless of the type of objects received, telepot generically calls them "message" (with a lowercase "m"). A message's *flavor* depends on the underlying object:

- a Message object gives the flavor `chat`
- a CallbackQuery object gives the flavor `callback_query`
- there are two more flavors, which you will come to shortly.

Use *`telepot.flavor()`* to check a message's flavor.

Here is a bot which does two things:

- When you send it a message, it gives you an inline keyboard.
- When you press a button on the inline keyboard, it says "Got it".

Pay attention to these things in the code:

- How I use namedtuple to construct an InlineKeyboardMarkup and an InlineKeyboardButton object
- *`telepot.glance()`* works on any type of messages. Just give it the flavor.
- Use *`Bot.answerCallbackQuery()`* to react to callback query
- To *route* messages according to flavor, give a *routing table* to *`Bot.message_loop()`*

```
import sys
import time
import telepot
from telepot.namedtuple import InlineKeyboardMarkup, InlineKeyboardButton

def on_chat_message(msg):
    content_type, chat_type, chat_id = telepot.glance(msg)

    keyboard = InlineKeyboardMarkup(inline_keyboard=[
                    [InlineKeyboardButton(text='Press me', callback_data='press')],
                ])

    bot.sendMessage(chat_id, 'Use inline keyboard', reply_markup=keyboard)

def on_callback_query(msg):
    query_id, from_id, query_data = telepot.glance(msg, flavor='callback_query')
    print('Callback Query:', query_id, from_id, query_data)

    bot.answerCallbackQuery(query_id, text='Got it')
```

```
TOKEN = sys.argv[1]  # get token from command-line

bot = telepot.Bot(TOKEN)
bot.message_loop({'chat': on_chat_message,
                  'callback_query': on_callback_query})
print('Listening ...')

while 1:
    time.sleep(10)
```

CHAPTER 11

Inline Query

So far, the bot has been operating in a chat - private, group, or channel.

In a private chat, Alice talks to Bot. Simple enough.

In a group chat, Alice, Bot, and Charlie share the same group. As the humans gossip in the group, Bot hears selected messages (depending on whether in privacy mode or not) and may chime in once in a while.

Inline query is a totally different mode of operations.

Imagine this. Alice wants to recommend a restaurant to Zach, but she can't remember the location right off her head. *Inside the chat screen with Zach*, Alice types `@Bot where is my favorite restaurant`, issuing an inline query to Bot, like asking Bot a question. Bot gives back a list of answers; Alice can choose one of them - as she taps on an answer, that answer is sent to Zach as a chat message. In this case, Bot never takes part in the conversation. Instead, *Bot acts as an assistant*, ready to give you talking materials. For every answer Alice chooses, Bot gets notified with a *chosen inline result*.

To enable a bot to receive InlineQuery, you have to send a `/setinline` command to BotFather. **An InlineQuery message gives the flavor** `inline_query`.

To enable a bot to receive ChosenInlineResult, you have to send a `/setinlinefeedback` command to BotFather. **A ChosenInlineResult message gives the flavor** `chosen_inline_result`.

In this code sample, pay attention to these things:

- How I use namedtuple InlineQueryResultArticle and InputTextMessageContent to construct an answer to inline query.
- Use *`Bot.answerInlineQuery()`* to send back answers

```
import sys
import telepot
from telepot.namedtuple import InlineQueryResultArticle, InputTextMessageContent

def on_inline_query(msg):
    query_id, from_id, query_string = telepot.glance(msg, flavor='inline_query')
    print ('Inline Query:', query_id, from_id, query_string)

    articles = [InlineQueryResultArticle(
                    id='abc',
                    title='ABC',
                    input_message_content=InputTextMessageContent(
                        message_text='Hello'
                    )
                )]
```

```
    bot.answerInlineQuery(query_id, articles)

def on_chosen_inline_result(msg):
    result_id, from_id, query_string = telepot.glance(msg, flavor='chosen_inline_result')
    print ('Chosen Inline Result:', result_id, from_id, query_string)

TOKEN = sys.argv[1]  # get token from command-line

bot = telepot.Bot(TOKEN)
bot.message_loop({'inline_query': on_inline_query,
                  'chosen_inline_result': on_chosen_inline_result},
                 run_forever='Listening ...')
```

However, this has a small problem. As you types and pauses, types and pauses, types and pauses ... closely bunched inline queries arrive. In fact, a new inline query often arrives *before* we finish processing a preceding one. With only a single thread of execution, we can only process the closely bunched inline queries sequentially. Ideally, whenever we see a new inline query coming from the same user, it should override and cancel any preceding inline queries being processed (that belong to the same user).

My solution is this. An *Answerer* takes an inline query, inspects its `from id` (the originating user id), and checks to see whether that user has an *unfinished* thread processing a preceding inline query. If there is, the unfinished thread will be cancelled before a new thread is spawned to process the latest inline query. In other words, an *Answerer* ensures **at most one** active inline-query-processing thread per user.

Answerer also frees you from having to call *Bot.answerInlineQuery()* every time. You supply it with a *compute function*. It takes that function's returned value and calls *Bot.answerInlineQuery()* to send the results. Being accessible by multiple threads, the compute function must be **thread-safe**.

```
import sys
import telepot
from telepot.namedtuple import InlineQueryResultArticle, InputTextMessageContent

def on_inline_query(msg):
    def compute():
        query_id, from_id, query_string = telepot.glance(msg, flavor='inline_query')
        print('Inline Query:', query_id, from_id, query_string)

        articles = [InlineQueryResultArticle(
                        id='abc',
                        title=query_string,
                        input_message_content=InputTextMessageContent(
                            message_text=query_string
                        )
                    )]

        return articles

    answerer.answer(msg, compute)

def on_chosen_inline_result(msg):
    result_id, from_id, query_string = telepot.glance(msg, flavor='chosen_inline_result')
    print ('Chosen Inline Result:', result_id, from_id, query_string)

TOKEN = sys.argv[1]  # get token from command-line

bot = telepot.Bot(TOKEN)
answerer = telepot.helper.Answerer(bot)
```

```
bot.message_loop({'inline_query': on_inline_query,
                  'chosen_inline_result': on_chosen_inline_result},
                 run_forever='Listening ...')
```

CHAPTER 12

Maintain Threads of Conversation

So far, we have been using a single line of execution to handle messages. That is adequate for simple programs. For more sophisticated programs where states need to be maintained across messages, a better approach is needed.

Consider this scenario. A bot wants to have an intelligent conversation with a lot of users, and if we could only use a single line of execution to handle messages (like what we have done so far), we would have to maintain some state variables about each conversation *outside* the message-handling function(s). On receiving each message, we first have to check whether the user already has a conversation started, and if so, what we have been talking about. To avoid such mundaneness, we need a structured way to maintain "threads" of conversation.

Let's look at my solution. Here, I implemented a bot that counts how many messages have been sent by an individual user. If no message is received after 10 seconds, it starts over (timeout). The counting is done *per chat* - that's the important point.

```
import sys
import telepot
from telepot.delegate import pave_event_space, per_chat_id, create_open

class MessageCounter(telepot.helper.ChatHandler):
    def __init__(self, *args, **kwargs):
        super(MessageCounter, self).__init__(*args, **kwargs)
        self._count = 0

    def on_chat_message(self, msg):
        self._count += 1
        self.sender.sendMessage(self._count)

TOKEN = sys.argv[1]  # get token from command-line

bot = telepot.DelegatorBot(TOKEN, [
    pave_event_space()(
        per_chat_id(), create_open, MessageCounter, timeout=10),
])
bot.message_loop(run_forever='Listening ...')
```

A *DelegatorBot* is able to spawn *delegates*. Above, it is spawning one `MessageCounter` *per chat id*.

Also noteworthy is *pave_event_space()*. To kill itself after 10 seconds of inactivity, the delegate schedules a timeout event. For events to work, we need to prepare an *event space*.

Detailed explanation of the delegation mechanism (e.g. how and when a `MessageCounter` is created, and why) is beyond the scope here. Please refer to *DelegatorBot*.

CHAPTER 13

Inline Handler per User

You may also want to answer inline query differently depending on user. When Alice asks Bot "Where is my favorite restaurant?", Bot should give a different answer than when Charlie asks the same question.

In the code sample below, pay attention to these things:

- *AnswererMixin* adds an *Answerer* instance to the object
- *per_inline_from_id()* ensures one instance of `QueryCounter` per originating user

```
import sys
import telepot
from telepot.delegate import pave_event_space, per_inline_from_id, create_open
from telepot.namedtuple import InlineQueryResultArticle, InputTextMessageContent

class QueryCounter(telepot.helper.InlineUserHandler, telepot.helper.AnswererMixin):
    def __init__(self, *args, **kwargs):
        super(QueryCounter, self).__init__(*args, **kwargs)
        self._count = 0

    def on_inline_query(self, msg):
        def compute():
            query_id, from_id, query_string = telepot.glance(msg, flavor='inline_query')
            print(self.id, ':', 'Inline Query:', query_id, from_id, query_string)

            self._count += 1
            text = '%d. %s' % (self._count, query_string)

            articles = [InlineQueryResultArticle(
                            id='abc',
                            title=text,
                            input_message_content=InputTextMessageContent(
                                message_text=text
                            )
                        )]

            return articles

        self.answerer.answer(msg, compute)

    def on_chosen_inline_result(self, msg):
        result_id, from_id, query_string = telepot.glance(msg, flavor='chosen_inline_result')
        print(self.id, ':', 'Chosen Inline Result:', result_id, from_id, query_string)

TOKEN = sys.argv[1]  # get token from command-line
```

```
bot = telepot.DelegatorBot(TOKEN, [
    pave_event_space()(
        per_inline_from_id(), create_open, QueryCounter, timeout=10),
])
bot.message_loop(run_forever='Listening ...')
```

CHAPTER 14

Async Version (Python 3.5+)

Everything discussed so far assumes traditional Python. That is, network operations are blocking; if you want to serve many users at the same time, some kind of threads are usually needed. Another option is to use an asynchronous or event-driven framework, such as Twisted.

Python 3.5 has its own `asyncio` module. Telepot supports that, too. If your bot is to serve many people, I strongly recommend doing it asynchronously.

If your O/S does not have Python 3.5 built in, you have to compile it yourself:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install libssl-dev openssl libreadline-dev
$ cd ~
$ wget https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
$ tar zxf Python-3.5.2.tgz
$ cd Python-3.5.2
$ ./configure
$ make
$ sudo make install
```

Finally:

```
$ pip3.5 install telepot
```

In case you are not familiar with asynchronous programming, let's start by learning about generators and coroutines:

- 'yield' and Generators Explained
- Sequences and Coroutines

... why we want asynchronous programming:

- Problem: Threads Are Bad

... how generators and coroutines are applied to asynchronous programming:

- Understanding Asynchronous IO
- A Curious Course on Coroutines and Concurrency

... and how an asyncio program is generally structured:

- The New asyncio Module in Python 3.4
- Event loop examples
- HTTP server and client

Telepot's async version basically mirrors the traditional version. Main differences are:

- blocking methods are now coroutines, and should be called with `await`
- delegation is achieved by tasks, instead of threads

Because of that (and this is true of asynchronous Python in general), a lot of methods will not work in the interactive Python interpreter like regular functions would. They will have to be driven by an event loop.

Async version is under module `telepot.aio`. I duplicate the message counter example below in async style:

- Substitute async version of selected classes and functions
- Use `async/await` to do asynchronous operations

```
import sys
import asyncio
import telepot
from telepot.aio.delegate import pave_event_space, per_chat_id, create_open

class MessageCounter(telepot.aio.helper.ChatHandler):
    def __init__(self, *args, **kwargs):
        super(MessageCounter, self).__init__(*args, **kwargs)
        self._count = 0

    async def on_chat_message(self, msg):
        self._count += 1
        await self.sender.sendMessage(self._count)

TOKEN = sys.argv[1]  # get token from command-line

bot = telepot.aio.DelegatorBot(TOKEN, [
    pave_event_space()(
        per_chat_id(), create_open, MessageCounter, timeout=10),
])

loop = asyncio.get_event_loop()
loop.create_task(bot.message_loop())
print('Listening ...')

loop.run_forever()
```

More Examples »

t

telepot.delegate, 7
telepot.exception, 17
telepot.routing, 19

A

Administrator (class in telepot.helper), 11
administrator (telepot.helper.ChatContext attribute), 10
answer() (telepot.helper.Answerer method), 16
answerCallbackQuery() (telepot.Bot method), 3
Answerer (class in telepot.helper), 16
answerer (telepot.helper.AnswererMixin attribute), 16
AnswererMixin (class in telepot.helper), 16
answerInlineQuery() (telepot.Bot method), 3
augment_bot() (telepot.helper.CallbackQueryCoordinator method), 15
augment_edit() (telepot.helper.CallbackQueryCoordinator method), 15
augment_on_close() (telepot.helper.IdleEventCoordinator method), 14
augment_on_message() (telepot.helper.CallbackQueryCoordinator method), 15
augment_on_message() (telepot.helper.IdleEventCoordinator method), 14
augment_send() (telepot.helper.CallbackQueryCoordinator method), 15

B

BadFlavor, 17
BadHTTPResponse, 17
Bot (class in telepot), 1
bot (telepot.helper.ListenerContext attribute), 10
BotWasBlockedError, 17
BotWasKickedError, 17
by_chat_command() (in module telepot.routing), 19
by_command() (in module telepot.routing), 19
by_content_type() (in module telepot.routing), 19
by_data() (in module telepot.routing), 20
by_regex() (in module telepot.routing), 20
by_text() (in module telepot.routing), 20

C

call() (in module telepot.delegate), 8
callback_query_coordinator (telepot.helper.InterceptCallbackQueryMixin attribute), 16
CallbackQueryCoordinator (class in telepot.helper), 14
CallbackQueryOriginContext (class in telepot.helper), 10
CallbackQueryOriginHandler (class in telepot.helper), 10
cancel() (telepot.helper.StandardEventScheduler method), 14
capture() (telepot.helper.Listener method), 12
capture_origin() (telepot.helper.CallbackQueryCoordinator method), 15
chain() (in module telepot.delegate), 9
chat_id (telepot.helper.ChatContext attribute), 10
ChatContext (class in telepot.helper), 10
ChatHandler (class in telepot.helper), 10
configure() (telepot.helper.CallbackQueryCoordinator method), 15
configure() (telepot.helper.StandardEventScheduler method), 13
create_open() (in module telepot.delegate), 9
create_run() (in module telepot.delegate), 8

D

DefaultRouterMixin (class in telepot.helper), 13
DelegatorBot (class in telepot), 6
deleteWebhook() (telepot.Bot method), 3
description (telepot.exception.TelegramError attribute), 17
DESCRIPTION_PATTERNS (telepot.exception.BotWasBlockedError attribute), 18
DESCRIPTION_PATTERNS (telepot.exception.BotWasKickedError attribute), 17
DESCRIPTION_PATTERNS (telepot.exception.MigratedToSupergroupChatError attribute), 18
DESCRIPTION_PATTERNS (tele-

pot.exception.TooManyRequestsError attribute), 18
DESCRIPTION_PATTERNS (telepot.exception.UnauthorizedError attribute), 17
download_file() (telepot.Bot method), 4

E

editMessageCaption() (telepot.Bot method), 3
editMessageReplyMarkup() (telepot.Bot method), 3
editMessageText() (telepot.Bot method), 3
Editor (class in telepot.helper), 11
editor (telepot.helper.CallbackQueryOriginContext attribute), 11
error_code (telepot.exception.TelegramError attribute), 17
event (telepot.exception.EventNotFound attribute), 17
event_at() (telepot.helper.StandardEventScheduler method), 13
event_later() (telepot.helper.StandardEventScheduler method), 14
event_now() (telepot.helper.StandardEventScheduler method), 14
event_space (telepot.helper.StandardEventScheduler attribute), 13
EventNotFound, 17

F

flance() (in module telepot), 5
flavor() (in module telepot), 5
fleece() (in module telepot), 6
forwardMessage() (telepot.Bot method), 1

G

getChat() (telepot.Bot method), 3
getChatAdministrators() (telepot.Bot method), 3
getChatMember() (telepot.Bot method), 3
getChatMembersCount() (telepot.Bot method), 3
getFile() (telepot.Bot method), 2
getGameHighScores() (telepot.Bot method), 4
getMe() (telepot.Bot method), 1
getUpdates() (telepot.Bot method), 3
getUserProfilePhotos() (telepot.Bot method), 2
getWebhookInfo() (telepot.Bot method), 3
glance() (in module telepot), 5

I

id (telepot.helper.ListenerContext attribute), 10
idle_event_coordinator (telepot.helper.IdleTerminateMixin attribute), 14
IdleEventCoordinator (class in telepot.helper), 14
IdleTerminate, 17
IdleTerminateMixin (class in telepot.helper), 14
include_callback_query_chat_id() (in module telepot.delegate), 9
InlineUserHandler (class in telepot.helper), 10
intercept_callback_query_origin() (in module telepot.delegate), 9
InterceptCallbackQueryMixin (class in telepot.helper), 15
is_event() (in module telepot), 6

J

json (telepot.exception.TelegramError attribute), 17

K

kickChatMember() (telepot.Bot method), 3

L

leaveChat() (telepot.Bot method), 3
Listener (class in telepot.helper), 12
listener (telepot.helper.ListenerContext attribute), 10
ListenerContext (class in telepot.helper), 10
lower_key() (in module telepot.routing), 20

M

make_content_type_routing_table() (in module telepot.routing), 21
make_event_data() (telepot.helper.StandardEventScheduler method), 13
make_routing_table() (in module telepot.routing), 20
map() (telepot.helper.Router method), 13
message_identifier() (in module telepot), 6
message_loop() (telepot.Bot method), 4
MigratedToSupergroupChatError, 18
Monitor (class in telepot.helper), 9

O

offender (telepot.exception.BadFlavor attribute), 17
on__idle() (telepot.helper.IdleTerminateMixin method), 14
on_message() (telepot.helper.DefaultRouterMixin method), 13
openable() (in module telepot.helper), 16
origin (telepot.helper.CallbackQueryOriginContext attribute), 11
origin_identifier() (in module telepot), 6

P

pair() (in module telepot.delegate), 9
pave_event_space() (in module telepot.delegate), 9
peel() (in module telepot), 6
per_application() (in module telepot.delegate), 8
per_callback_query_chat_id() (in module telepot.delegate), 8

per_callback_query_origin() (in module telepot.delegate), 8
per_chat_id() (in module telepot.delegate), 7
per_chat_id_except() (in module telepot.delegate), 7
per_chat_id_in() (in module telepot.delegate), 7
per_event_source_id() (in module telepot.delegate), 8
per_from_id() (in module telepot.delegate), 7
per_from_id_except() (in module telepot.delegate), 7
per_from_id_in() (in module telepot.delegate), 7
per_inline_from_id() (in module telepot.delegate), 8
per_inline_from_id_except() (in module telepot.delegate), 8
per_inline_from_id_in() (in module telepot.delegate), 8
per_message() (in module telepot.delegate), 8
process_key() (in module telepot.routing), 20

R

refresh() (telepot.helper.IdleEventCoordinator method), 14
response (telepot.exception.BadHTTPResponse attribute), 17
route() (telepot.helper.Router method), 13
Router (class in telepot.helper), 12
router (telepot.helper.DefaultRouterMixin attribute), 13

S

SafeDict (class in telepot.helper), 16
scheduler (telepot.helper.StandardEventMixin attribute), 14
seconds (telepot.exception.WaitTooLong attribute), 17
sendAudio() (telepot.Bot method), 2
sendChatAction() (telepot.Bot method), 2
sendContact() (telepot.Bot method), 2
sendDocument() (telepot.Bot method), 2
Sender (class in telepot.helper), 11
sender (telepot.helper.ChatContext attribute), 10
sender (telepot.helper.UserContext attribute), 10
sendGame() (telepot.Bot method), 2
sendLocation() (telepot.Bot method), 2
sendMessage() (telepot.Bot method), 1
sendPhoto() (telepot.Bot method), 2
sendSticker() (telepot.Bot method), 2
sendVenue() (telepot.Bot method), 2
sendVideo() (telepot.Bot method), 2
sendVoice() (telepot.Bot method), 2
setGameScore() (telepot.Bot method), 3
setWebhook() (telepot.Bot method), 3
StandardEventMixin (class in telepot.helper), 14
StandardEventScheduler (class in telepot.helper), 13
status (telepot.exception.BadHTTPResponse attribute), 17
StopListening, 17

T

TelegramError, 17
telepot.delegate (module), 7
telepot.exception (module), 17
telepot.routing (module), 19
TelepotException, 17
text (telepot.exception.BadHTTPResponse attribute), 17
TooManyRequestsError, 18

U

UnauthorizedError, 17
unbanChatMember() (telepot.Bot method), 3
uncapture_origin() (telepot.helper.CallbackQueryCoordinator method), 15
until() (in module telepot.delegate), 9
upper_key() (in module telepot.routing), 20
user_id (telepot.helper.UserContext attribute), 10
UserContext (class in telepot.helper), 10
UserHandler (class in telepot.helper), 10

W

wait() (telepot.helper.Listener method), 12
WaitTooLong, 17