Telepot Tutorial

Installation

pip:

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

easy_install:

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

Download manually:

```
$ wget https://pypi.python.org/packages/source/t/telepot/telepot-8.2.zip
$ unzip telepot-8.2.zip
$ cd telepot-8.2
$ python setup.py install
```

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.

Test the account

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

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)
[]
```

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.

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!')
```

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`.

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 sent 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**.

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` or `edited_chat` (because the sender may edit a previous message)
- a CallbackQuery object gives the flavor `callback_query`
- and there are 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)
```

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 ...')
```

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 per_chat_id, create_open

class MessageCounter(telepot.helper.ChatHandler):
    def __init__(self, seed_tuple, timeout):
        super(MessageCounter, self).__init__(seed_tuple, timeout)
        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, [
    (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*.

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 **`telepot.DelegatorBot`**.

Per-User Inline Handler

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 per_inline_from_id, create_open
from telepot.namedtuple import InlineQueryResultArticle, InputTextMessageContent

class QueryCounter(telepot.helper.InlineUserHandler, telepot.helper.AnswererMixin):
    def __init__(self, seed_tuple, timeout):
        super(QueryCounter, self).__init__(seed_tuple, timeout)
        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, [
    (per_inline_from_id(), create_open(QueryCounter, timeout=10)),
])
bot.message_loop(run_forever='Listening ...')
```

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:

```
$ sudo 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. Pay attention to these things:

- 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 per_chat_id, create_open

class MessageCounter(telepot.aio.helper.ChatHandler):
    def __init__(self, seed_tuple, timeout):
        super(MessageCounter, self).__init__(seed_tuple, timeout)
        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, [
    (per_chat_id(), create_open(MessageCounter, timeout=10)),
])

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

loop.run_forever()
```

Usage

I am composing a page illustrating common usages. It is coming soon ...

Reference

Traditional Version
Async Version