Writing a Telegram Antispam Bot in Python:

An introduction to async programming

FOSDEM 2023 – 05.02.2023

Brussels, Belgium

Marc-André Lemburg :: eGenix.com GmbH
Speaker Introduction

• Marc-André Lemburg
  – Python since 1994
  – Studied Mathematics
  – CEO eGenix.com GmbH
  – Consulting CTO, Senior Solutions Architect and Coach
  – EuroPython Society Fellow and former Chair
  – Python Software Foundation Fellow
  – Python Core Developer
  – Based in Düsseldorf, Germany

  – More and for connecting: http://malemburg.com
Motivation: Writing a Telegram Antispam Bot

• Our Python Meeting Düsseldorf user group has been getting lots of signup spam since early last year
  − No longer possible to handle those manually

• Issues
  − Link Spam
  − Crypto Spam
  − Shady Offers
  − Scraping of contact infos
Solution: Write a low resource, scalable TG bot

- Use a scalable Python library for writing Telegram Bots:

  **pyrogram**
  - Fairly new library, actively maintained
  - **Fully asynchronous**
  - Uses the Telegram API directly (without proxy)
  - Open Source: LGPL 3
But what’s this “asynchronous programming”?

Let’s have a look at different execution models…

- **Synchronous** execution
- **Threaded** execution
- **Asynchronous** execution
Terminology: Synchronous / Threaded / Asynchronous

• **Synchronous**
  – All instructions are executed one after another
  – I/O and similar external resources cause execution to wait
  – **Timing is not a problem. Everything is deterministic.**
  – **Problem: Waiting is not an efficient use of resources :-)**
Terminology: Synchronous / Threaded / Asynchronous

- **Threaded**
  - Several synchronous parts of the program run in parallel, using OS threads
  - Execution is controlled by the OS, not the application
  - Threads are often assigned to different CPU cores
  - **Problem**: Sequence of execution is not necessarily deterministic
  - **Problem**: Unexpected delays can happen
  - **Problem**: Sharing data is hard – requires locks
  - **Problem**: OS overhead
  - **Advantage**: Efficient use of resources
Terminology: Synchronous / Threaded / Asynchronous

• **Asynchronous**
  
  – While some parts of the program wait for e.g. I/O, other parts can continue to run
  – Execution is controlled by the application, not the OS
  – This is not the same as “running in parallel“ (threading)
  
  – **Problem:** Sequence of execution is not necessarily deterministic
  – **Problem:** Unexpected delays can happen
  – **Problem:** Scope limited to a single core
  – **Problem:** All parts of the code have to collaborate
  
  – **Advantage:** Efficient use of resources
Python: Global Interpreter Lock (GIL)

- **The GIL makes sure that only one thread runs Python byte code at any point in time**
  - Only released for I/O or other long running tasks...
  - ... and then only if no Python code can be run

- **Threads can only share the Python Interpreter, not use it simultaneously**
  - Result: Even if you have multiple cores in the CPUs, only one thread can run Python byte code
  - All other threads which want to run Python code have to wait
Python: Threaded code on multiple cores/threads

- **Threaded + multiple cores/threads:**
  - A lot of waiting
    - Threads need to wait for the GIL
      - Delays due to I/O
    - Not much parallel work
      - (mostly only while doing I/O)
Python: Threaded code – a closer look
Python: Asynchronous to saturate a single core/thread

- **Asynchronous with one thread/process:**
  - Less waiting
  - All application parts have to participate
  - Active passing of control (cooperative)
  - Less overhead compared to threads
  - No parallel work, only simulated
  - More efficient use of a single core
Asynchronous programming in Python

- **Coroutines**
  - Like “subroutines”, but routine can internally give up control to the calling function where needed
  - Created by calling an async function in Python

- **New keywords in Python 3.5+**
  - Make working with coroutines a lot easier
  - `async def` task() - defines a coroutine
  - `await an_io_call()` - gives up control, until `an_io_call()` responds

- **Package asyncio**
  - Provides the `event loop` to run coroutines
  - Many other helpers to run coroutines
async + await: Example

Synchronous

```
1 import asyncio
2 import time
3 
4 # Synchronous
5 def task_sync(x):
6     print ('Task sync: {x} working')
7     time.sleep(2)
8     print ('Task sync: {x} done')
9 
10 task_sync('Example 1')
11 
12 print ('-'*72)
```

Asynchronous

```
14 # Asynchronous
15 async def task_async(x):
16     print (f'Task async: {x} working')
17     await asyncio.sleep(2)
18     print (f'Task async: {x} done')
19 
20 # Call task
21 tasks = (task_async('Example 2'),
22          task_async('Example 3'),
23          )
24 
25 async def main():
26     await asyncio.gather(*tasks)
27     asyncio.run(main())
```
async + await: Blocking calls / Giving up control

Synchronous

```python
import asyncio
import time

# Synchronous
def task_sync(x):
    print(f'Task sync: {x} working')
    time.sleep(2)
    print(f'Task sync: {x} done')
task_sync('Example 1')
print('-'*72)
```

Asynchronous

```python
# Asynchronous
async def task_async(x):
    print(f'Task async: {x} working')
    await asyncio.sleep(2)
    print(f'Task async: {x} done')

tasks = (task_async('Example 2'),
         task_async('Example 3'))
async def main():
    await asyncio.gather(*tasks)
asyncio.run(main())
```
async + await: Running sync / async functions

Synchronous

```python
import asyncio
import time

# Synchronous
def task_sync(x):
    print(f'Task sync: {x} working')
    time.sleep(2)
    print(f'Task sync: {x} done')
task_sync('Example 1')
print('-'*72)
```

Asynchronous

```python
# Asynchronous
async def task_async(x):
    print(f'Task async: {x} working')
    await asyncio.sleep(2)
    print(f'Task async: {x} done')

# Call task
tasks = (task_async('Example 2'),
         task_async('Example 3'),
         )
async def main():
    await asyncio.gather(*tasks)
asyncio.run(main())
```
The asyncio module: a closer look

• **Management functions for coroutines**
  – asyncio.run() – runs a coroutine immediately *(in a new event loop)*
  – asyncio.gather() – runs multiple coroutines (as tasks) in parallel and waits for completion of all of them
  – asyncio.sleep() – sleep for coroutines *(let’s other coroutines run)*

• **Waiting on coroutines**
  – asyncio.wait_for() – wait for a coroutine *(with timeout)*
  – asyncio.wait() – wait for a set of tasks/coroutines *(with timeout)*
The asyncio module: a closer look

- **Task objects**
  - Represents a scheduled coroutine call
  - Run by the event loop
  - `asyncio.Task` – task object type *(don’t create directly)*
  - `Task.cancel()` – cancels a Task object
  - `Task.done()` – returns True, iff the coroutine has been called
  - etc.

- **Scheduling tasks / coroutines**
  - `asyncio.create_task()` – create and schedule a Task object
  - `asyncio.current_task()` – returns the currently running task object
  - `asyncio.all_task()` – returns all task objects
Running async: the Event Loop

• Task objects are run by an event loop
  – Tasks run until the next await is hit
    Processing then goes back to the event loop
  – There can only be one event loop per thread
  – asyncio.get_running_loop() returns the loop object

• Blocking code
  – Examples: loading data with non-async code, long running calculation
  – It is possible to run blocking code in a separate thread
    to not have it block the event loop:
    • loop.run_in_executor()
    • asyncio.to_thread() (Python 3.9+)
• There are lots of other features and tools available in the asyncio world:
  - Subprocesses
  - Exceptions
  - Servers
  - Timers
  - Signal handlers
  - Sockets with async support
  - File descriptors with async support
  - Different event loop types
  - etc.

```python
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

**asyncio** — Asynchronous I/O

**Hello World!**

```python
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio is a library to write concurrent code using the `async/await` syntax.

asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

asyncio is often a perfect fit for IO-bound and high-level structured network code.
Async eco system: Lowest level

• **Python Standard Lib**
  – asyncio

• **Event Loops**
  – Event loop implementations often come with integrations for sockets, streams, files, pipes, DNS, network connections, etc.
  – asyncio.loop – Standard event loop
  – uvloop – Faster loop variant for asyncio using libuv

• **Alternative stacks**
  – Trio – Alternative async library, making things a bit easier / more concise
  – AnyIO – Abstraction for asyncio and trio
Async eco system: Low level

- **AIO Libs**
  - Collection of many async packages for Python’s asyncio
  - https://github.com/aio-libs

- **Examples**
  - aiohttp – HTTP client / server
  - aiopg – PostgreSQL interface
  - aiomysql – MySQL interface
  - aioredis – Redis interface
  - aiodns – DNS client
  - *(lots more)*

*Warning:* The database packages often don’t support transactions!
Async eco system: High level

• Web
  – ASGI – Async variant of WSGI
  – Tornado – Web framework
  – Starlette – New ASGI web framework
  – Quart – Async web framework similar to Flask
  – Django 3.0 – Django is starting to support ASGI as well
  – Uvicorn – ASGI server (similar to gunicorn for WSGI)

• APIs
  – FastAPI – REST API server
  – Tartiflette – GraphQL server
  – Strawberry – GraphQL server
Let’s apply this new knowledge...

... in the Telegram Antispam Bot:

https://github.com/egenix/egenix-telegram-antispam-bot

or search for “egenix telegram”
Implementation of the Bot

- **Subclassing of pyrogram’s Client**
  - Configuration via a Python config.py
    - Use os.environ for overrides
  - Delivered as a Python package
    - Easy to install
    - Provide __main__.py, to make python -m package work
- **Observability**
  - Use logging for simple debugging
  - Send admin messages to an admin Telegram group for easy monitoring
Don’t use Bot commands – process all messages

- Use the catch all handler
- Delegate tasks to other methods
- Where I/O happens, use async

```python
# Handles

async def all_messages(self, client, message):
    
    """ Handler which receives all messages sent to the chat.  
    This delegates the handling to other methods.  
    """
    if _debug:
        self.log('New message:', message)
    if not self.check_access(message):
        return

    # Ignore messages without a .from_user attribute
    if not message.from_user:
        return

    member_id = message.from_user.id

    # Ignore messages sent by the bot itself
    if member_id == self.bot_id:
        return

    # Delegate some messages to other handlers:
    if message.new_chat_members:
        # Process new chat members message
        return await self.new_chat_members(client, message)
```
Async works almost like sync code ...

... with just a few `await` added, meaning: “wait for an answer”

```python
async def welcome_new_member(self, message):
    
    """ Accept and welcome the user as a new member to the group. 
    This concludes the conversation and removes the member from 
    the .new_members dict. 
    """
    message needs to point to the user's signup message.
    
    chat_id = message.chat.id
    new_member = message.new_member
    await self.remove_conversation(message)
    await self.send_message(
        chat_id,
        f'Thank you for answering the welcome question, '
        f'{new_member.first_name}. '
        f'You are now a member of the chat. '
        f'Please introduce yourself to the group in a line or two.')
    self.new_members.pop(new_member.id)
    await self.log_admin(
        f'Accepted application by '
        f'[{new_member.first_name}]
        f'(username={new_member.username}, id={new_member.id})]
        f'(tg://user?id={new_member.id})'
    )
```
Antispam Bot: Results

• Since end of April 2022, the bot banned 780+ spam signups until today
  – Saved more than around 26 hours of admin work
  – Break even reached

• Saved us from an unknown number of spam messages

• Mission accomplished
Main takeaway: Async is great – give it a try!

These are exciting times
Thank you for your attention!

Time for discussion
Contact

eGenix.com Software, Skills and Services GmbH
Marc-André Lemburg
Pastor-Löh-Str. 48
D-40764 Langenfeld
Germany

eMail: mal@egenix.com
Phone: +49 211 9304112
Fax: +49 211 3005250
Web: https://www.egenix.com/

LinkedIn:
References

• Several photos taken from Pixabay
• Some screenshots taken from the mentioned websites
• All other graphics and photos are (c) eGenix.com or used with permission
• Details are available on request
• Logos are trademarks of their resp. trademark holders