Asyncio Documentation Documentation

Release 0.0

Victor Stinner
Mike Müller

Mar 13, 2018

Contents

CHAPTER 1

Chapter 1: First steps with asyncio

1.1 Why use asyncio?

1.1.1 Why asynchronous programming?

asyncio is a library to write asynchronous applications. It is the most efficient way to implement a network server having to handle many concurrent users.

1.1.2 But gevent and eventlet just work!

Or *Why should I bother with all these extra annoying async and await keywords?*.

In short, asyncio adopted a radically different solution for race conditions.

Parallel computing using threads is hard because of race conditions. Gevent and eventlet have a similar issue using "green" (lightweight) threads.

Code written with asyncio is less error-prone: by just looking at the code, it is possible to identify which parts of the code are under our controls and where the event loop takes over the control flow and is able to run other tasks when our task is waiting for something.

gevent and eventlet are designed to hide the asynchronous programming. For non-expert, and sometimes even for experts, it is really hard to guess where the event loop is allowed to suspend the task and run other tasks in background. It is even worse. A modification in a third party library can change the behaviour of our code, introduce a new point where the task is suspended.

For an example, see the "Ca(sh|che Coherent) Money" section of the Unyielding article (by Glyph, February, 2014).

1.2 Getting Started

1.2.1 Python 3.5 (or higher) only

This documentation is written for Python 3.5 to avail of the new `async` and `await` keywords.

If you have Python 3.5 installed you only need to install `aiohttp`:

```
pip install -U aiohttp
```

If you don't have Python 3.5 installed yet, you have several options to install it.

All platforms with `conda`

- Download and install Miniconda for our platform.
- Create a new Python 3.5 environment (named `aio35`, use a different if you like):

  ```
  conda create -n aio35 python=3.5
  ```

- Activate it. Linux and OS X:

  ```
  $ source activate aio35
  ```

  Windows:

  ```
  $ source activate aio35
  ```

- Install `aiohttp`:

  ```
  $(aio35) pip install aiohttp
  ```

Platform specific

- Windows: The easiest way to use Python 3.5 would be to use a package manager such as conda. See the installation instructions above.
- Mac OS X: Install Homebrew and then type `brew install python3`
- Linux: Ubuntu 16.04+ and Arch linux ship with Python 3.5 included. If you don't have Python 3.5+ on your computer, you can compile it or use Pythonz.

1.2.2 Create a virtual environment to run examples

If you don't use conda (see above), create a virtual environment:

```
python3 -m venv venv
```

Note: Depending on your platform, the Python 3 interpreter could be invoked by `python` instead. This is the case for conda on Windows for example.

Install `aiohttp` in the virtual environment:

```
./venv/bin/python -m pip install -U aiohttp
```

1.3 Hello World

This is a series of examples showing the basics of how to write *coroutines* and schedule them in the asyncio *event loop*.

1.3.1 Simple coroutine

This example uses the `asyncio.BaseEventLoop.run_until_complete()` method to schedule a simple function that will wait one second, print `hello` and then finish.

Because it is launched with `run_until_complete()`, the *event loop* itself will terminate once the *coroutine* is completed.

```
import asyncio

async def say(what, when):
    await asyncio.sleep(when)
    print(what)

loop = asyncio.get_event_loop()
loop.run_until_complete(say('hello world', 1))
loop.close()
```

1.3.2 Creating tasks

This second example shows how you can schedule multiple *coroutines* in the event loop, and then run the *event loop*.

Notice that this example will print `second_hello` before `first_hello`, as the first *task* scheduled waits longer that the second one before printing.

Also note that this example will never terminate, as the *loop* is asked to *run_forever*.

```
import asyncio

async def say(what, when):
    await asyncio.sleep(when)
    print(what)


loop = asyncio.get_event_loop()

loop.create_task(say('first hello', 2))
loop.create_task(say('second hello', 1))

loop.run_forever()
loop.close()
```

1.3.3 Stopping the loop

This third example adds another *task* that will stop the *event loop* before all scheduled *tasks* could execute, which results in a warning.

```
import asyncio

async def say(what, when):
    await asyncio.sleep(when)
    print(what)

async def stop_after(loop, when):
    await asyncio.sleep(when)
    loop.stop()


loop = asyncio.get_event_loop()

loop.create_task(say('first hello', 2))
loop.create_task(say('second hello', 1))
loop.create_task(say('third hello', 4))
loop.create_task(stop_after(loop, 3))

loop.run_forever()
loop.close()
```

Warning:

```
Task was destroyed but it is pending!
task: <Task pending coro=<say() done, defined at examples/loop_stop.py:3>
wait_for=<Future pending cb=[Task._wakeup()]>>
```

1.4 Hello Clock

Example illustrating how to schedule two *coroutines* to run concurrently. They run for ten minutes, during which the first *coroutine* is scheduled to run every second, while the second is scheduled to run every minute.

The function *asyncio.gather* is used to schedule both *coroutines* at once.

```
import asyncio


async def print_every_second():
    "Print seconds"
    while True:
        for i in range(60):
            print(i, 's')
            await asyncio.sleep(1)


async def print_every_minute():
    for i in range(1, 10):
        await asyncio.sleep(60)
        print(i, 'minute')
```

```
loop = asyncio.get_event_loop()
loop.run_until_complete(
    asyncio.gather(print_every_second(),
                   print_every_minute())
)
loop.close()
```

1.5 HTTP client example

HTTP client example:

```
import asyncio
import aiohttp

async def fetch_page(session, url):
    with aiohttp.Timeout(10):
        async with session.get(url) as response:
            assert response.status == 200
            return await response.read()

loop = asyncio.get_event_loop()
with aiohttp.ClientSession(loop=loop) as session:
    content = loop.run_until_complete(
        fetch_page(session, 'http://python.org'))
    print(content)
loop.close()
```

For more information, see aiohttp documentation.

1.6 asyncio performance

Random notes about tuning asyncio for performance. Performance means two different terms which might be incompatible:

- Number of concurrent requests per second
- Request latency in seconds: min/average/max time to complete a request

1.6.1 Architecture: Worker processes

Because of its GIL, CPython is basically only able to use 1 CPU. To increase the number of concurrent requests, one solution is to spawn multiple worker processes. See for example:

- Gunicorn
- API-Hour

1.6.2 Stream limits

- limit parameter of StreamReader/open_connection()
- set_write_buffer_limits() low/high water mark on writing for transports

aiohttp uses `set_writer_buffer_limits(0)` for backpressure support and implemented their own buffering, see:

- aio-libs/aiohttp#1369
- Some thoughts on asynchronous API design in a post-async/await world (November, 2016) by Nathaniel J. Smith

1.6.3 TCP_NODELAY

Since Python 3.6, asyncio now sets the `TCP_NODELAY` option on newly created sockets: disable the Nagle algorithm for send coalescing. Disable segment buffering so data can be sent out to peer as quickly as possible, so this is typically used to improve network utilisation.

See Nagle's algorithm.

1.6.4 TCP_QUICKACK

(This option is not used by asyncio by default.)

The `TCP_QUICKACK` option can be used to send out acknowledgements as early as possible than delayed under some protocol level exchanging, and it's not stable/permanent, subsequent TCP transactions (which may happen under the hood) can disregard this option depending on actual protocol level processing or any actual disagreements between user setting and stack behaviour.

1.6.5 Tune the Linux kernel

Linux TCP sysctls:

- `/proc/sys/net/ipv4/tcp_mem`
- `/proc/sys/net/core/rmem_default` and `/proc/sys/net/core/rmem_max`: The default and maximum amount for the receive socket memory
- `/proc/sys/net/core/wmem_default` and `/proc/sys/net/core/wmem_max`: The default and maximum amount for the send socket memory
- `/proc/sys/net/core/optmem_max`: The maximum amount of option memory buffers
- `net.ipv4.tcp_no_metrics_save`
- `net.core.netdev_max_backlog`: Set maximum number of packets, queued on the INPUT side, when the interface receives packets faster than kernel can process them.

1.7 Learn asyncio if you come from Twisted

The Twisted project is probably one of the oldest libraries that supports asynchronous programming in Python. It has been used by many programmers to develop a variety of applications. It supports many network protocols and can be used for many different types of network programming. In fact, asyncio was heavily inspired by Twisted. The expertise of several Twisted developers had been incorporated in asyncio. Soon, there will be a version of Twisted that is based on asyncio.

1.7.1 Rosetta Stone

This tables shows equivalent concepts in Twisted and asyncio.

Twisted	asyncio
`Deferred`	`asyncio.Future`
`deferToThread(func)`	`loop.run_in_executor(None, func)`
`@inlineCallbacks`	`async def`
`reactor.run()`	`loop.run_forever()`

1.7.2 Deferred example

This small example shows two equivalent programs, one implemented in Twisted and one in asyncio.

Twisted

Basic Twisted example using deferred:

```
from twisted.internet import defer
from twisted.internet import reactor


def multiply(x):
    result = x * 2
    d = defer.Deferred()
    reactor.callLater(1.0, d.callback,
                      result)
    return d


def step1(x):
    return multiply(x)


def step2(result):
    print("result: %s" % result)

    reactor.stop()


d = defer.Deferred()
d.addCallback(step1)
d.addCallback(step2)
d.callback(5)

reactor.run()
```

asyncio

Similar example written using asyncio:

```
import asyncio


async def multiply(x):
    result = x * 2
    await asyncio.sleep(1)
    return result


async def steps(x):
    result = await multiply(x)
    print("result: %s" % result)


loop = asyncio.get_event_loop()
coro = steps(5)
loop.run_until_complete(coro)
loop.close()
```

1.8 Getting Help

1.8.1 Mailing list

- async-sig: discussions about asynchronous programming in Python (i.e. async/await)

- python-tulip Google Group: historical name of the official asyncio mailing list

1.8.2 StackOverflow

There is an python-asyncio tag on StackOverflow where you can read new questions but also read answers of previous questions.

1.8.3 IRC

There is an IRC channel `#asyncio` on the Freenode server.

CHAPTER 2

Chapter 2: Advanced topics

2.1 TCP echo client and server

2.1.1 TCP echo client

TCP echo client using streams:

```
import asyncio


async def tcp_echo_client(message, loop):
    reader, writer = await asyncio.open_connection('127.0.0.1', 8888,
                                                   loop=loop)

    print('Send: %r' % message)
    writer.write(message.encode())

    data = await reader.read(100)
    print('Received: %r' % data.decode())

    print('Close the socket')
    writer.close()


message = 'Hello World!'
loop = asyncio.get_event_loop()
loop.run_until_complete(tcp_echo_client(message, loop))
loop.close()
```

2.1.2 TCP echo server

TCP echo server using streams:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')
    print("Received %r from %r" % (message, addr))

    print("Send: %r" % message)
    writer.write(data)
    await writer.drain()

    print("Close the client socket")
    writer.close()

loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo, '127.0.0.1', 8888, loop=loop)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

2.2 Threads

Run slow CPU-intensive or blocking I/O code in a thread:

```
import asyncio


def compute_pi(digits):
    # implementation
    return 3.14


async def main(loop):
    digits = await loop.run_in_executor(None, compute_pi, 20000)
    print("pi: %s" % digits)


loop = asyncio.get_event_loop()
loop.run_until_complete(main(loop))
loop.close()
```

See also:

- run_in_executor() documentation

- asyncio: Concurrency and multithreading

2.3 Subprocess

2.3.1 Run a subprocess and read its output

A simple example to run commands in a subprocess using asyncio.create_subprocess_exec and get the output using process.communicate:

```
import asyncio


async def run_command(*args):
    # Create subprocess
    process = await asyncio.create_subprocess_exec(
        *args,
        # stdout must a pipe to be accessible as process.stdout
        stdout=asyncio.subprocess.PIPE)
    # Wait for the subprocess to finish
    stdout, stderr = await process.communicate()
    # Return stdout
    return stdout.decode().strip()


loop = asyncio.get_event_loop()
# Gather uname and date commands
commands = asyncio.gather(run_command('uname'), run_command('date'))
# Run the commands
uname, date = loop.run_until_complete(commands)
# Print a report
print('uname: {}, date: {}'.format(uname, date))
loop.close()
```

2.3.2 Communicate with a subprocess using standard streams

A simple example to communicate with an echo subprocess using process.stdin and process.stdout:

```
import asyncio


async def echo(msg):
    # Run an echo subprocess
    process = await asyncio.create_subprocess_exec(
        'cat',
        # stdin must a pipe to be accessible as process.stdin
        stdin=asyncio.subprocess.PIPE,
        # stdout must a pipe to be accessible as process.stdout
        stdout=asyncio.subprocess.PIPE)
    # Write message
    print('Writing {!r} ...'.format(msg))
    process.stdin.write(msg.encode() + b'\n')
    # Read reply
    data = await process.stdout.readline()
    reply = data.decode().strip()
```

```
    print('Received {!r}'.format(reply))
    # Stop the subprocess
    process.terminate()
    code = await process.wait()
    print('Terminated with code {}'.format(code))


loop = asyncio.get_event_loop()
loop.run_until_complete(echo('hello!'))
loop.close()
```

For more information, see the asyncio subprocess documentation.

2.4 Producer/consumer

2.4.1 Simple example

A simple producer/consumer example, using an asyncio.Queue:

```
import asyncio
import random


async def produce(queue, n):
    for x in range(1, n + 1):
        # produce an item
        print('producing {}/{}'.format(x, n))
        # simulate i/o operation using sleep
        await asyncio.sleep(random.random())
        item = str(x)
        # put the item in the queue
        await queue.put(item)

    # indicate the producer is done
    await queue.put(None)


async def consume(queue):
    while True:
        # wait for an item from the producer
        item = await queue.get()
        if item is None:
            # the producer emits None to indicate that it is done
            break

        # process the item
        print('consuming item {}...'.format(item))
        # simulate i/o operation using sleep
        await asyncio.sleep(random.random())


loop = asyncio.get_event_loop()
queue = asyncio.Queue(loop=loop)
producer_coro = produce(queue, 10)
consumer_coro = consume(queue)
```

```
loop.run_until_complete(asyncio.gather(producer_coro, consumer_coro))
loop.close()
```

2.4.2 Using task_done()

A simple producer/consumer example, using Queue.task_done and Queue.join:

```
import asyncio
import random


async def produce(queue, n):
    for x in range(n):
        # produce an item
        print('producing {}/{}'.format(x, n))
        # simulate i/o operation using sleep
        await asyncio.sleep(random.random())
        item = str(x)
        # put the item in the queue
        await queue.put(item)


async def consume(queue):
    while True:
        # wait for an item from the producer
        item = await queue.get()

        # process the item
        print('consuming {}...'.format(item))
        # simulate i/o operation using sleep
        await asyncio.sleep(random.random())

        # Notify the queue that the item has been processed
        queue.task_done()


async def run(n):
    queue = asyncio.Queue()
    # schedule the consumer
    consumer = asyncio.ensure_future(consume(queue))
    # run the producer and wait for completion
    await produce(queue, n)
    # wait until the consumer has processed all items
    await queue.join()
    # the consumer is still awaiting for an item, cancel it
    consumer.cancel()


loop = asyncio.get_event_loop()
loop.run_until_complete(run(10))
loop.close()
```

For more information, see the asyncio queue documentation.

2.5 Asyncio Debug Mode

- Set the environment variable: `PYTHONASYNCIODEBUG=1`
- Configure Python logging: `logging.basicConfig(level=logging.ERROR)`

See also: Debug mode of asyncio (ref doc).

CHAPTER 3

Chapter 3: Larger examples

3.1 Web Scraping

Web scraping means downloading multiple web pages, often from different servers. Typically, there is a considerable waiting time between sending a request and receiving the answer. Using a client that always waits for the server to answer before sending the next request, can lead to spending most of time waiting. Here `asyncio` can help to send many requests without waiting for a response and collecting the answers later. The following examples show how a synchronous client spends most of the time waiting and how to use `asyncio` to write asynchronous client that can handle many requests concurrently.

3.1.1 A Mock Web Server

This is a very simple web server. (See below for the code.) Its only purpose is to wait for a given amount of time. Test it by running it from the command line:

```
$ python simple_server.py
```

It will answer like this:

```
Serving from port 8000 ...
```

Now, open a browser and go to this URL:

```
http://localhost:8000/
```

You should see this text in your browser:

```
Waited for 0.00 seconds.
```

Now, add `2.5` to the URL:

```
http://localhost:8000/2.5
```

After pressing enter, it will take 2.5 seconds until you see this response:

```
Waited for 2.50 seconds.
```

Use different numbers and see how long it takes until the server responds.

The full implementation looks like this:

```
# file: simple_server.py

"""Simple HTTP server with GET that waits for given seconds.
"""

from http.server import BaseHTTPRequestHandler, HTTPServer
from socketserver import ThreadingMixIn
import time


ENCODING = 'utf-8'


class ThreadingHTTPServer(ThreadingMixIn, HTTPServer):
    """Simple multi-threaded HTTP server.
    """
    pass


class MyRequestHandler(BaseHTTPRequestHandler):
    """Very simple request handler. Only supports GET.
    """

    def do_GET(self):  # pylint: disable=invalid-name
        """Respond after seconds given in path.
        """
        try:
            seconds = float(self.path[1:])
        except ValueError:
            seconds = 0.0
        if seconds < 0:
            seconds = 0.0
        text = "Waited for {:4.2f} seconds.\nThat's all.\n"
        msg = text.format(seconds).encode(ENCODING)
        time.sleep(seconds)
        self.send_response(200)
        self.send_header("Content-type", 'text/plain; charset=utf-8')
        self.send_header("Content-length", str(len(msg)))
        self.end_headers()
        self.wfile.write(msg)


def run(server_class=ThreadingHTTPServer,
        handler_class=MyRequestHandler,
        port=8000):
    """Run the simple server on given port.
    """
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    print('Serving from port {} ...'.format(port))
    httpd.serve_forever()
```

```
if __name__ == '__main__':
    run()
```

Let's have a look into the details. This provides a simple multi-threaded web server:

```
class ThreadingHTTPServer(ThreadingMixIn, HTTPServer):
    """Simple multi-threaded HTTP server.
    """
    pass
```

It uses multiple inheritance. The mix-in class `ThreadingMixIn` provides the multi-threading support and the class `HTTPServer` a basic HTTP server.

The request handler only has a `GET` method:

```
class MyRequestHandler(BaseHTTPRequestHandler):
    """Very simple request handler. Only supports GET.
    """

    def do_GET(self):  # pylint: disable=invalid-name
        """Respond after seconds given in path.
        """
        try:
            seconds = float(self.path[1:])
        except ValueError:
            seconds = 0.0
        if seconds < 0:
            seconds = 0.0
        text = "Waited for {:4.2f} seconds.\nThat's all.\n"
        msg = text.format(seconds).encode(ENCODING)
        time.sleep(seconds)
        self.send_response(200)
        self.send_header("Content-type", 'text/plain; charset=utf-8')
        self.send_header("Content-length", str(len(msg)))
        self.end_headers()
        self.wfile.write(msg)
```

It takes the last entry in the paths with `self.path[1:]`, i.e. our `2.5`, and tries to convert it into a floating point number. This will be the time the function is going to sleep, using `time.sleep()`. This means waiting 2.5 seconds until it answers. The rest of the method contains the HTTP header and message.

3.1.2 A Synchronous Client

Our first attempt is synchronous. This is the full implementation:

```
"""Synchronous client to retrieve web pages.
"""
```

```
from urllib.request import urlopen
import time

ENCODING = 'ISO-8859-1'


def get_encoding(http_response):
    """Find out encoding.
    """
    content_type = http_response.getheader('Content-type')
    for entry in content_type.split(';'):
        if entry.strip().startswith('charset'):
            return entry.split('=')[1].strip()
    return ENCODING


def get_page(host, port, wait=0):
    """Get one page supplying `wait` time.

    The path will be build with: `host:port/wait`
    """
    full_url = '{}:{}/{}'.format(host, port, wait)
    with urlopen(full_url) as http_response:
        html = http_response.read().decode(get_encoding(http_response))
    return html


def get_multiple_pages(host, port, waits, show_time=True):
    """Get multiple pages.
    """
    start = time.perf_counter()
    pages = [get_page(host, port, wait) for wait in waits]
    duration = time.perf_counter() - start
    sum_waits = sum(waits)
    if show_time:
        msg = 'It took {:4.2f} seconds for a total waiting time of {:4.2f}.'
        print(msg.format(duration, sum_waits))
    return pages


if __name__ == '__main__':

    def main():
        """Test it.
        """
        pages = get_multiple_pages(host='http://localhost', port='8000',
                                   waits=[1, 5, 3, 2])
        for page in pages:
            print(page)

    main()
```

Again, we go through it step-by-step.

While about 80 % of the websites use `utf-8` as encoding (provided by the default in `ENCODING`), it is a good idea to actually use the encoding specified by `charset`. This is our helper to find out what the encoding of the page is:

```
def get_encoding(http_response):
    """Find out encoding.
    """
    content_type = http_response.getheader('Content-type')
    for entry in content_type.split(';'):
        if entry.strip().startswith('charset'):
            return entry.split('=')[1].strip()
    return ENCODING
```

It falls back to `ISO-8859-1` if it cannot find a specification of the encoding.

Using `urllib.request.urlopen()`, retrieving a web page is rather simple. The response is a bytestring and `.encode()` is needed to convert it into a string:

```
def get_page(host, port, wait=0):
    """Get one page supplying `wait` time.

    The path will be build with: `host:port/wait`
    """
    full_url = '{}:{}/{}'.format(host, port, wait)
    with urlopen(full_url) as http_response:
        html = http_response.read().decode(get_encoding(http_response))
    return html
```

Now, we want multiple pages:

```
def get_multiple_pages(host, port, waits, show_time=True):
    """Get multiple pages.
    """
    start = time.perf_counter()
    pages = [get_page(host, port, wait) for wait in waits]
    duration = time.perf_counter() - start
    sum_waits = sum(waits)
    if show_time:
        msg = 'It took {:4.2f} seconds for a total waiting time of {:4.2f}.'
        print(msg.format(duration, sum_waits))
    return pages
```

We just iterate over the waiting times and call `get_page()` for all of them. The function `time.perf_counter()` provides a time stamp. Taking two time stamps a different points in time and calculating their difference provides the elapsed run time.

Finally, we can run our client:

```
$ python synchronous_client.py
```

and get this output:

```
It took 11.08 seconds for a total waiting time of 11.00.
Waited for 1.00 seconds.
That's all.

Waited for 5.00 seconds.
That's all.

Waited for 3.00 seconds.
That's all.

Waited for 2.00 seconds.
That's all.
```

Because we wait for each call to `get_page()` to complete, we need to wait about 11 seconds. That is the sum of all waiting times. Let's see if we can do it any better going asynchronously.

3.1.3 Getting One Page Asynchronously

This module contains a functions that reads a page asynchronously, using the new Python 3.5 keywords `async` and `await`:

```
# file: async_page.py

"""Get a "web page" asynchronously.
"""

import asyncio

ENCODING = 'ISO-8859-1'


def get_encoding(header):
    """Find out encoding.
    """
    for line in header:
        if line.lstrip().startswith('Content-type'):
            for entry in line.split(';'):
                if entry.strip().startswith('charset'):
                    return entry.split('=')[1].strip()
    return ENCODING


async def get_page(host, port, wait=0):
    """Get a "web page" asynchronously.
    """
    reader, writer = await asyncio.open_connection(host, port)
    writer.write(b'\r\n'.join([
        'GET /{} HTTP/1.0'.format(wait).encode(ENCODING),
        b'Host: %b' % host.encode(ENCODING),
        b'Connection: close',
        b'', b''
    ]))
    header = []
    msg_lines = []
    async for raw_line in reader:
        line = raw_line.decode(ENCODING).strip()
```

```
        if not line.strip():
            break
        header.append(line)
    encoding = get_encoding(header)
    async for raw_line in reader:
        line = raw_line.decode(encoding).strip()
        msg_lines.append(line)
    writer.close()
    return '\n'.join(msg_lines)
```

As with the synchronous example, finding out the encoding of the page is a good idea. This function helps here by going through the lines of the HTTP header, which it gets as an argument, searching for `charset` and returning its value if found. Again, the default encoding is `ISO-8859-1`:

```
def get_encoding(header):
    """Find out encoding.
    """
    for line in header:
        if line.lstrip().startswith('Content-type'):
            for entry in line.split(';'):
                if entry.strip().startswith('charset'):
                    return entry.split('=')[1].strip()
    return ENCODING
```

The next function is way more interesting because it actually works asynchronously:

```
async def get_page(host, port, wait=0):
    """Get a "web page" asynchronously.
    """
    reader, writer = await asyncio.open_connection(host, port)
    writer.write(b'\r\n'.join([
        'GET /{} HTTP/1.0'.format(wait).encode(ENCODING),
        b'Host: %b' % host.encode(ENCODING),
        b'Connection: close',
        b'', b''
    ]))
    header = []
    msg_lines = []
    async for raw_line in reader:
        line = raw_line.decode(ENCODING).strip()
        if not line.strip():
            break
        header.append(line)
    encoding = get_encoding(header)
    async for raw_line in reader:
        line = raw_line.decode(encoding).strip()
        msg_lines.append(line)
    writer.close()
    return '\n'.join(msg_lines)
```

The function `asyncio.open_connection()` opens a connection to the given URL. It returns a coroutine. Using `await`, which had to be `yield from` in Python versions prior to 3.5, it yields an instance of a `StreamReader`

and one of a `StreamWriter`. These only work within the event loop.

Now, we can send a `GET` request, suppling our waiting time by writing to the `StreamWriter` instance `writer`. The request has to be in bytes. Therefore, we need to convert our strings in to bytestrings.

Next, we read header and message from the reader, which is a `StreamReader` instance. We need to iterate over the reader by using a special or loop for `asyncio`:

```
async for raw_line in reader:
```

Header and message are dived by an empty line. We just stop the iteration as soon as we found an empty line. Handing the header over too `get_encoding()` provides the encoding of the retrieved page. The `.decode()` method uses this encoding to convert the read bytes into strings. After closing the writer, we can return the message lines joined by newline characters.

3.1.4 Getting Multiple Pages Asynchronously - Without Time Savings

This is our first approach retrieving multiple pages, using our asynchronous `get_page()`:

```
"""Get "web pages.

Waiting until one pages is download before getting the next."
"""

import asyncio
from contextlib import closing
import time

from async_page import get_page


def get_multiple_pages(host, port, waits, show_time=True):
    """Get multiple pages.
    """
    start = time.perf_counter()
    pages = []
    with closing(asyncio.get_event_loop()) as loop:
        for wait in waits:
            pages.append(loop.run_until_complete(get_page(host, port, wait)))
    duration = time.perf_counter() - start
    sum_waits = sum(waits)
    if show_time:
        msg = 'It took {:4.2f} seconds for a total waiting time of {:4.2f}.'
        print(msg.format(duration, sum_waits))
    return pages

if __name__ == '__main__':

    def main():
        """Test it.
        """
        pages = get_multiple_pages(host='localhost', port='8000',
                                   waits=[1, 5, 3, 2])
        for page in pages:
            print(page)

    main()
```

The interesting things happen in a few lines in `get_multiple_pages()` (the rest of this function just measures the run time and displays it):

```
    with closing(asyncio.get_event_loop()) as loop:
        for wait in waits:
            pages.append(loop.run_until_complete(get_page(host, port, wait)))
```

The `closing` from the standard library module `contextlib` starts the event loop within a context and closes the loop when leaving the context:

```
with closing(asyncio.get_event_loop()) as loop:
    <body>
```

The two lines above are equivalent to these five lines:

```
loop = asyncio.get_event_loop():
try:
    <body>
finally:
    loop.close()
```

We call `get_page()` for each page in a loop. Here we decide to wrap each call in `loop.run_until_complete()`:

```
for wait in waits:
    pages.append(loop.run_until_complete(get_page(host, port, wait)))
```

This means, we wait until each pages has been retrieved before asking for the next. Let's run it from the command-line to see what happens:

```
$ async_client_blocking.py
It took 11.06 seconds for a total waiting time of 11.00.
Waited for 1.00 seconds.
That's all.
Waited for 5.00 seconds.
That's all.
Waited for 3.00 seconds.
That's all.
Waited for 2.00 seconds.
That's all.
```

So it still takes about eleven seconds in total. We made it more complex and did not improve speed. Let's see if we can do better.

3.1.5 Getting Multiple Pages Asynchronously - With Time Savings

We want to take advantage of the asynchronous nature of `get_page()` and save time. We modify our client to use a list with four instances of a *task*. This allows us to send out requests for all pages we want to retrieve without waiting for the answer before asking for the next page:

```
"""Get "web pages.

Waiting until one pages is download before getting the next."
"""

import asyncio
```

```
from contextlib import closing
import time

from async_page import get_page


def get_multiple_pages(host, port, waits, show_time=True):
    """Get multiple pages.
    """
    start = time.perf_counter()
    pages = []
    tasks = []
    with closing(asyncio.get_event_loop()) as loop:
        for wait in waits:
            tasks.append(get_page(host, port, wait))
        pages = loop.run_until_complete(asyncio.gather(*tasks))
    duration = time.perf_counter() - start
    sum_waits = sum(waits)
    if show_time:
        msg = 'It took {:4.2f} seconds for a total waiting time of {:4.2f}.'
        print(msg.format(duration, sum_waits))
    return pages

if __name__ == '__main__':

    def main():
        """Test it.
        """
        pages = get_multiple_pages(host='localhost', port='8000',
                                   waits=[1, 5, 3, 2])
        for page in pages:
            print(page)

    main()
```

The interesting part is in this loop:

```
with closing(asyncio.get_event_loop()) as loop:
    for wait in waits:
        tasks.append(get_page(host, port, wait))
    pages = loop.run_until_complete(asyncio.gather(*tasks))
```

We append all return values of `get_page()` to our lits of tasks. This allows us to send out all request, in our case four, without waiting for the answers. After sending all of them, we wait for the answers, using:

```
loop.run_until_complete(asyncio.gather(*tasks))
```

We used `loop.run_until_complete()` already for each call to `get_page()` in the previous section. The difference here is the use of `asyncio.gather()` that is called with all our tasks in the list `tasks` as arguments. The `asyncio.gather(*tasks)` means for our example with four list entries:

```
asyncio.gather(tasks[0], tasks[1], tasks[2], tasks[3])
```

So, for a list with 100 tasks it would mean:

```
asyncio.gather(tasks[0], tasks[1], tasks[2],
               # 96 more tasks here
               tasks[99])
```

Let's see if we got any faster:

```
$ async_client_nonblocking.py
It took 5.08 seconds for a total waiting time of 11.00.
Waited for 1.00 seconds.
That's all.
Waited for 5.00 seconds.
That's all.
Waited for 3.00 seconds.
That's all.
Waited for 2.00 seconds.
That's all.
```

Yes! It works. The total run time is about five seconds. This is the run time for the longest wait. Now, we don't have to wait for the sum of `waits` but rather for `max(waits)`.

We did quite a bit of work, sending a request and scanning an answer, including finding out the encoding. There should be a shorter way as these steps seem to be always necessary for getting the page content with the right encoding. Therefore, in the next section, we will have a look at high-level library `aiohttp` that can help to make our code shorter.

Exercise

Add more waiting times to the list `waits` and see how this impacts the run times of the blocking and the non-blocking implementation. Try (positive) numbers that are all less than five. Then try numbers greater than five.

3.1.6 High-Level Approach with `aiohttp`

The library aiohttp allows to write HTTP client and server applications, using a high-level approach. Install with:

```
$ pip install aiohttp
```

The whole program looks like this:

```
"""aiohttp-based client to retrieve web pages.
"""

import asyncio
from contextlib import closing
import time

import aiohttp


async def fetch_page(session, host, port=8000, wait=0):
    """Get one page.
    """
    url = '{}:{}/{}'.format(host, port, wait)
    with aiohttp.Timeout(10):
        async with session.get(url) as response:
            assert response.status == 200
            return await response.text()
```

```
def get_multiple_pages(host, waits, port=8000, show_time=True):
    """Get multiple pages.
    """
    tasks = []
    pages = []
    start = time.perf_counter()
    with closing(asyncio.get_event_loop()) as loop:
        with aiohttp.ClientSession(loop=loop) as session:
            for wait in waits:
                tasks.append(fetch_page(session, host, port, wait))
            pages = loop.run_until_complete(asyncio.gather(*tasks))
    duration = time.perf_counter() - start
    sum_waits = sum(waits)
    if show_time:
        msg = 'It took {:4.2f} seconds for a total waiting time of {:4.2f}.'
        print(msg.format(duration, sum_waits))
    return pages


if __name__ == '__main__':

    def main():
        """Test it.
        """
        pages = get_multiple_pages(host='http://localhost', port='8000',
                                   waits=[1, 5, 3, 2])
        for page in pages:
            print(page)

    main()
```

The function to get one page is asynchronous, because of the `async def`:

```
async def fetch_page(session, host, port=8000, wait=0):
    """Get one page.
    """
    url = '{}:{}/{}'.format(host, port, wait)
    with aiohttp.Timeout(10):
        async with session.get(url) as response:
            assert response.status == 200
            return await response.text()
```

The arguments are the same as those for the previous function to retrieve one page plus the additional argument `session`. The first task is to construct the full URL as a string from the given host, port, and the desired waiting time.

We use a timeout of 10 seconds. If it takes longer than the given time to retrieve a page, the programm throws a `TimeoutError`. Therefore, to make this more robust, you might want to catch this error and handle it appropriately.

The `async with` provides a context manager that gives us a response. After checking the status being `200`, which means that all is alright, we need to `await` again to return the body of the page, using the method `text()` on the response.

This is the interesting part of `get_multiple_pages()`:

```
with closing(asyncio.get_event_loop()) as loop:
    with aiohttp.ClientSession(loop=loop) as session:
        for wait in waits:
            tasks.append(fetch_page(session, host, port, wait))
        pages = loop.run_until_complete(asyncio.gather(*tasks))
```

It is very similar to the code in the example of the time-saving implementation with `asyncio`. The only difference is the opened client session and handing over this session to `fetch_page()` as the first argument.

Finally, we run this program:

```
$ python aiohttp_client.py
It took 5.04 seconds for a total waiting time of 11.00.
Waited for 1.00 seconds.
That's all.

Waited for 5.00 seconds.
That's all.

Waited for 3.00 seconds.
That's all.

Waited for 2.00 seconds.
That's all.
```

It also takes about five seconds and gives the same output as our version before. But the implementation for getting a single page is much simpler and takes care of the encoding and other aspects not mentioned here.

CHAPTER 4

Indices and tables

4.1 Glossary

coroutine A coroutine is a piece of code that can be paused and resumed. In contrast to threads which are preemptively multitasked by the operating system, coroutines multitask cooperatively. I.e. they choose when to pause (or to use terminology for coroutines before 3.4 - `yield`) execution. They can also execute other coroutines.

event loop The event loop is the central execution device to launch execution of coroutines and handle I/O (Network, sub-processes...)

future It's like a mailbox where you can subscribe to receive a result when it will be done. More details in official documentation

task It represents the execution of a coroutine and take care the result in a future. More details in official documentation

CHAPTER 5

See also

- asyncio wiki
- asyncio Reference Documentation.
- A Web Crawler With asyncio Coroutines by A. Jesse Jiryu Davis and Guido van Rossum
- Writing Redis in Python with asyncio: Part 1 by James Saryerwinnie

CHAPTER 6

Contributing

6.1 Asyncio documentation

- Online doc: https://asyncio.readthedocs.io/
- GitHub: https://github.com/asyncio-doc/asyncio-doc
- AsyncIO documentation is written with Sphinx.

6.2 Notes to writers

Tutorials should use Python 3.5 `async` and `await` keywords rather than `@asyncio.coroutine` and `yield from`.

6.3 Ideas

- Advanced section:
  - protocols and transports: as least point to good implementations
  - explain how to *test* asyncio applications. Twisted documentation example

6.4 How to install Sphinx

Firstly, you need to install the Sphinx tool using the Linux package manager like apt-get or dnf for example.

But if you want to install it via pip , you can create a virtual environment with the *venv* module of Python 3

```
python3 -m venv env
source env/bin/activate
pip install -r requirements.txt
```

Once you have installed Sphinx, you can build the documentation.

6.5 How to build the documentation

Install Sphinx using the Linux package manager like apt-get or dnf for example. Then build the documentation using:

```
make html
```

6.6 See also

- https://github.com/python/asyncio
- http://krondo.com/an-introduction-to-asynchronous-programming-and-twisted/
- https://curio.readthedocs.io/en/latest/tutorial.html

Index