asks Documentation

Release 1.3.6

Mark Jameson

Apr 01, 2022

Contents

CHAPTER 1

Contents:

1.1 `asks` - An overview of the functions and kw/arguments.

`asks` is *heavily* influenced by requests, and as such pretty much everything that works in `requests` works in `asks`. So, if you're familiar with the format you can pretty much skip to the distinctions regarding sessions

The examples here use the base one-request-functions for verbosity's sake, but all of these functions are completely transferable to the `Session` class as methods.

Warning!

If you don't use a `Session` you can easily max out your OS's socket resources against highly performant servers (usually local to the machine or LAN). When using the base functions you'll be creating a new connection for every request.

(Calling `asks.get('https://some-url.io'))` really makes a temporary `Session`.)

1.1.1 General HTTP methods

`asks` supports `get()`, `head()`, `post()`, `put()`, `delete()`, `options()`, `patch()` and `request()`.

`request` takes a HTTP method as a string for its first argument.

When using the basic functions they each require a URI:

```
import asks

async def blah():
    a = await asks.get('https://example.com')
    s = await asks.head('http://httpbin.org')
    k = await asks.post('https://webservice.net:25000')
    s = await asks.put('www.your-coat-on.net') # <- no scheme! Will fail!
    # etc.
    r = await asks.request('GET', 'http://httpbin.org/get')
```

A scheme *must* be supplied. Port can be set by providing it in the URI.

All functions / methods share the same set of args / keyword args, though not all are appropriate for every HTTP method.

1.1.2 Passing Queries

The `params` and `data` args take a dictionary and convert it in to a query string to be appended to to URL, or sent in the request body, respectively.

```
async def example():
    r = await asks.get('www.example.com', params={'Elmo': 'wants data'})

# sends as request path:
b'?Elmo=wants+data'

async def example():
    r = await asks.get('www.example.com', data={'Elmo': 'wants data'})

# sends in request body:
b'Elmo=wants+data'
```

You may also pass strings, `asks` will attempt to format them correctly.

```
async def example():
    r = await asks.post('www.example.com', params='Elmo wants data')

# sends as request path:
b'?Elmo%20wants%20data'

async def example():
    r = await asks.post('www.example.com', data='Elmo wants data')

# sends in request body:
b'Elmo wants data'
```

Note: the `data` *arg is incompatible with the* `json`, `multipart` *and* `files` *args.*

1.1.3 Custom Headers

Add your own custom headers or overwrite the default headers by supplying your own dict to the `headers` argument. Note that user headers set in this way will, if conflicting, take precedence.

```
async def example():
    r = await asks.get('www.example.com',
                       headers={'Custom-Header': 'My value'})
```

1.1.4 Sending JSON

Pass Python `dict` objects to the `json` argument to send them as JSON in your request. Note that if your workflow here involves opening a JSON file, you should use `curio`'s `aopen()` or `trio`'s `open_file()` to avoid stalling the program on disk reads.

```
dict_to_send = {'Data_1': 'Important thing',
                'Data_2': 'Really important thing'}

async def example():
    r = await asks.post('www.example.com', json=dict_to_send)
```

Note: the `json` *arg is incompatible with the* `data`, `multipart` *and* `files` *args.*

1.1.5 Sending Files (multipart/form-data)

Pass a `dict` in the form `{field: value}` (as many as you like) to the `multipart` argument to send a `multipart/form-data` request.

To send files, send one of the following as `value`:

- A `pathlib.Path` object: `asks` will asyncronously open and read the file.
- An already open binary file-like object. The `read()` method can be a normal function or a coroutine (remember a normal file may block!). You can use `anyio.aopen` to get an async file object.
- An `asks.multipart.MultipartData` object, which can be used to override the filename or the mime-type of the sent file.

Other values are converted to strings and sent directly.

```
async def send_file():
    r = await asks.post('http://httpbin.org/post',
                        multipart={'file_1': Path('my_file.txt')})
    pprint(r.json())

# if we wanted to send both an already open file and some random data:
from anyio import aopen

async def send_file_and_data():
    async with await aopen('my_file.txt', 'rb') as my_file:
        r = await asks.post('http://httpbin.org/post',
                            multipart={'file_1': my_file,
                                       'some_data': 'I am multipart hear me roar',
                                       'some_integer': 3})

# if we wanted to send some bytes as a file:
from asks.multipart import MultipartData

async def send_bytes():
    r = await asks.post('http://httpbin.org/post',
                        multipart={'file_1':
                            MultipartData(b'some text',
                                          mime_type='text/plain',
                                          basename='my_file.txt')})
    pprint(r.json())

# if we wanted to override metadata:

async def send_customized_file():
    r = await asks.post('http://httpbin.org/post',
                        multipart={'file_1':
                            MultipartData(Path('my_file.txt'),
                                          mime_type='text/plain',
```

(continues on next page)

(continued from previous page)

```
                                                basename='some_other_name.txt')})
    pprint(r.json())
```

Note: the `multipart` *arg is incompatible with the* `data`, `json` *and* `files` *args.*

There is also the older `files` API, but `multipart` should be preferred over it. To use it, pass a `dict` in the form `{filename: filepath}` (as many as you like) and `asks` will asyncronously get the file data, building a multipart-formatted HTTP body. You can also pass non-file paths if you wish to send arbitrary multipart body data sections.

```
async def send_file():
    r = await asks.post('http://httpbin.org/post',
                        files={'file_1': 'my_file.txt'})

# if we wanted to send both a file and some random data:
async def send_file_and_data():
    r = await asks.post('http://httpbin.org/post',
                        files={'file_1': 'my_file.txt',
                               'some_data': 'I am multipart hear me roar'})
```

Note: the `files` *arg is incompatible with the* `data`, `json` *and* `multipart` *args.*

1.1.6 Sending Cookies

Pass a `dict` of cookie name(key) / value pairs to the `cookies` arg to ship 'em off.

```
async def example():
    r = await asks.get('www.example.com',
                       cookies={'Cookie Monster': 'Yum'})
```

1.1.7 Cookie Interactions

By default `asks` does not return sent cookies. To enable two-way cookie interactions, just pass `persist_cookies=True`.

```
async def example():
    r = await asks.get('www.example.com', persist_cookies=True)
```

1.1.8 Set Encoding

The default encoding is `utf-8`. You may override this by supplying a different encoding, be it a standard encoding or a custom one you've registered locally.

```
async def example():
    r = await asks.get('www.example.com', encoding='Latin-1')
```

Handy list of builtin encodings: https://gist.github.com/theelous3/7d6a3fe20a21966b809468fa336195e3

1.1.9 Limiting Redirects

You can limit the number of redirects by setting `max_redirects`. By default, the number of redirects is `20`. `asks` will not redirect on HEAD requests.

```
async def example():
    r = await asks.get('www.httpbin.org/redirect/3', max_redirects=2)
```

1.1.10 Preventing Redirects

You can prevent `asks` from automatically following redirects by setting `follow_redirects` to `False`. By default, `asks` will automatically follow redirects until a non-redirect response or `max_redirects` are encountered.

```
async def example():
    r = await asks.get('www.httpbin.org/redirect/3', follow_redirects=False)
```

1.1.11 Set Timeout(s)

Don't want to wait forever? Me neither. You may set a timeout with the `timeout` arg. This limits the time allotted for the request.

```
async def example():
    r = await asks.get('www.httpbin.org/redirect/3', timeout=1)
```

Note that the `timeout` arg does not account for the time required to actually establish the connection. That is controlled by a second timeout, the `connection_timeout`, which defaults to 60 seconds. It's used in the exact same way as `timeout`. For reasoning, read this.

There is a third timeout available for `StreamResponse.body` iteration. See The Response Object

1.1.12 Retry limiting

You can set a maximum number of retries with `retries`. This defaults to `1`, to catch sockets that die in the connection pool, or generally misbehave. There is no upper limit. Be careful :D

```
async def example():
    r = await asks.get('www.beat_dead_horses.org/neverworks', retries=9999999)
```

1.1.13 Authing

Available off the bat, we have HTTP basic auth and HTTP digest auth.

To add auth in asks, you pass a tuple of `('username', 'password')` to the `__init__` of an auth class. For example:

```
import asks
from asks import BasicAuth, DigestAuth

usr_pw = ('AzureDiamond', 'hunter2')

async def main():
```

(continues on next page)

(continued from previous page)

```
    r = await asks.get('https://some_protected.resource',
                       auth=BasicAuth(usr_pw))
    r2 = await asks.get('https://other_protected.thingy',
                       auth=DigestAuth(usr_pw),
                       auth_off_domain=True)
```

Note: `asks` will not pass auth along to connections that switch from HTTP to HTTPS, or off domain locations, unless you pass `auth_off_domain=True` to the call.

1.1.14 Streaming response data

You can stream the body of a response by setting `stream=True` , and iterating the response object's `.body` . An example of downloading a file:

```
import asks
import curio

async def main():
    r = await asks.get('http://httpbin.org/image/png', stream=True)
    async with curio.aopen('our_image.png', 'ab') as out_file:
        async for bytechunk in r.body:
            out_file.write(bytechunk)

curio.run(main())
```

It is important to note that if you do not iterate the `.body` to completion, bad things may happen as the connection sits there and isn't returned to the connection pool. You can get around this by context-managering the `.body` if there is a chance you might not iterate fully.

```
import asks
import curio

async def main():
    r = await asks.get('http://httpbin.com/image/png', stream=True)
    async with curio.aopen('our_image.png', 'wb') as out_file:
        async with r.body: # Bam! Safe!
            async for bytechunk in r.body:
                await out_file.write(bytechunk)

curio.run(main())
```

This way, once you leave the `async with` block, `asks` will automatically ensure the underlying socket is handled properly. You may also call `.body.close()` to manually close the stream.

The streaming body can also be used for streaming feeds and stuff of twitter and the like.

For some examples of how to use this, look here

1.1.15 Callbacks

Similar enough to streaming as seen above, but happens during the processing of the response body, before the response object is returned. Overall probably worse to use than streaming in every case but I'm sure someone will find a use for it.

The `callback` argument lets you pass a function as a callback that will be run on each bytechunk of response body *as the request is being processed*.

For some examples of how to use this, look here

1.2 `asks` - A Look at Sessions

While something like `requests` makes grabbing a single request very simple (and `asks` does too!), the `Session` in `asks` aim to make getting a great many things simple as well.

`asks`' `Session` methods are the same as the base `asks` functions, supporting `.get()`, `.head()`, `.post()`, `.put()`, `.delete()`, `.options()`, `.patch()` and `.request()`.

For more info on how to use these methods, take a look-see.

The `asks` `Session` has all of the features you would expect, and some extra that make working with web APIs a little nicer.

1.2.1 `Session` creation

To create a regular old session and start flinging requests down the pipe, do just that:

```
from asks import Session

async def main():
    s = Session()
    r = await s.get('https://example.org')
```

Well. That wasn't very exciting. Next, let's make a whole pile of requests, and modify the `connections` parameter.

1.2.2 !Important! Connection (un)limiting

The `Session`'s `connections` argument dictates the maximum number of concurrent connections `asks` will be allowed to make at any point during the `Sessions` lifespan. You *will* want to change the number of connections to a value that suits your needs and the server's limitations. If no data is publicly available to guide you here, err on the low side.

The default number of connections in the pool for a Session is a measly ONE. If I arbitrarily picked a number greater than one it would be too high for 49% of people and too low for the other 49%.

```
import asks
import curio

a_list_of_many_urls = ['wow', 'so', 'many', 'urls']

async def worker(s, url):
    r = await s.get(url)
    print(r.text)

async def main(url_list):
    s = asks.Session(connections=20)
    for url in url_list:
        await curio.spawn(worker(s, url))

curio.run(main(a_list_of_many_urls))
```

1.2.3 Session Headers

You can provide session-wide headers to your `requests` with the `headers` kwargument on `Session` instantiation, or by manually modifying the `.headers` attribute of your session.

```
from asks import Session

async def main():
    s = Session('https://example.com', headers={'Applies-to': 'all requests'})
    s.headers.update({'also-applies-to': 'all requests'})
```

If you send headers with a HTTP method's `headers` kwargument, it will take precedence. For example, in the above example; doing `s.get(headers={'Applies-to': 'this request only'})` will overwrite the session-wide header `'Applies-to'` for that single request.

1.2.4 Persistent Cookies

HTTP is stateless, and by default `asks` is too. You can turn stateful cookie-returning on by supplying the `persist_cookies=True` kwarg on session instantiation.

```
from asks import Session

async def main():
    s = Session('https://example.com', persist_cookies=True)
```

1.2.5 An alternate approach to web APIs

Often you'll want to programatically make many quite similar calls to a webservice. Worrying about constructing and reconstructing urls can be a pain, so `asks` has support for a different approach.

`Session` 's have a `base_location` and `endpoint` attribute which can be programatically set, and augmented using a HTTP method's `path` parameter.

In the next example, we'll make 1k calls over fifty connections to http://echo.jsontest.com. We'll do much of the same as above, except we'll set a base location of `http://echo.jsontest.com` an `endpoint` of `/asks/test` and in each request pass a number as a `path`, like `/1`.

The result will be a bunch of calls that look like

- `http://echo.jsontest.com/asks/test/1`
- `http://echo.jsontest.com/asks/test/2`
- `http://echo.jsontest.com/asks/test/etc.`

Please don't actually do this or the http://jsontest.com website will be very unhappy.

```
import asks
import curio

async def worker(s, num):
    r = await s.get(path='/' + str(num))
    print(r.text)

async def main():
    s = asks.Session(connections=50)
```

(continues on next page)

(continued from previous page)

```
    s.base_location = 'http://echo.jsontest.com'
    s.endpoint = '/asks/test'
    for i in range(1, 1001):
        await curio.spawn(worker(s, i))

curio.run(main())
```

You may override the `base_location` and `endpoint` by passing a URL normally.

1.3 `asks` - The Response Object

A plain ol' response object, `Response` is returned by default.

It has some attribs/properties to access the response content. Nothing too voodoo.

If you set `stream=True` a `StreamResponse` object is returned.

Both response types are subclasses of `BaseResponse`, for all of your typing needs.

1.3.1 Encoding

By default the `Response` object uses `utf-8`.

The response object will try to glean encoding from the response headers if available, before it's returned.

You can override the response-set or default encoding with either a built-in encoding or one you've registered locally with your `codecs` module by accessing the response's `.encoding` attribute.

```
async def main():
    r = await asks.get('http://example.com')
    r.encoding = 'latin-1'
```

1.3.2 Status Line

The three parts of the status line are the HTTP-Version, Status-Code and Reason-Phrase. They can be accessed as attributes of the response object like so:

```
async def main():
    r = await asks.get('http://example.com')

    r.http_version  # -> '1.1'
    r.status_code   # -> 200
    r.reason_phrase # -> 'OK'
```

1.3.3 Headers

The headers are available as a `dict` through `Response.headers`

```
async def main():
    r = await asks.get('http://example.com')
    print(r.headers)
```

(continues on next page)

(continued from previous page)

```
# Results in:
# {'Content-Encoding': 'gzip', 'Accept-Ranges': 'bytes', ...
```

1.3.4 JSON

Only available on `Response` objects.

If the response body is valid JSON you can load it as a Python `dict` by calling the response object's `.json()` method.

If the response was compressed, it will be decompressed.

```
async def main():
    r = await asks.get('http://httpbin.org/get')
    j = r.json()
    print(j)

# Results in
# {'args': {}, 'headers': {'Accept': '*/*', 'Accept-Encoding', ...
```

1.3.5 View Body (text decoded, content, raw)

These are only available on the `Response` object; returned when `stream=False`, which is the default behaviour.

Generally the way to see the body as it was intended is to use the `.content` property. This will return the content as is, after decompression if there was any.

For something slightly more human-readable, you may want to try the `.text` property. This will attempt to decompress (if needed) and decode the content (with `.encoding`). This for example, makes HTML and JSON etc. quite readable in your shell.

To view the body exactly as it was sent, just use the `.body` attribute. Note that this may be compressed madness, so don't worry if you can't read it with your poor wee eyes.

```
async def main():
    r = await asks.get('http://example.com')

    r.content
    r.text
    r.body
```

1.3.6 Streaming

If the request was made with `stream=True`, the object returned will be a `StreamResponse` whose `.body` attribute will point to an iterable `StreamBody` object from which you can stream data.

You may add a timeout to each poll for data by including `timeout` in the creation of the context manager. Example below alongside disabling data decompression.

To disable automatic decompression on the stream, set the `StreamBody.decompress_data` to `False`.

```
async def main():
    r = await asks.get('http://example.com')
    r.decompress_data = False
    async for chunk in r.body(timeout=5):
        print(r)
```

1.3.7 Cookies

Each response object will keep a list of any cookies set during the response, accessible by the `.cookies` attribute. Each cookie is a `Cookie` object. They are pretty basic. Here's a list of attributes:

- `.name`
- `.value`
- `.domain`
- `.path`
- `.secure`
- `.expires`
- `.comment`
- `.host`

There may be more values set by the response.

1.3.8 Response History

If any redirects or 401-requiring auth attempts were handled during the request, the response objects for those requests will be stored in the final response object's `.history` attribute in a list. Any response objects found in there are exactly like your main response object, and have all of the above methods, properties, and attributes.

```
async def main():
    r = await asks.get('http://httpbin.org/redirect/3')
    print(r.history)
    print(r.history[1].status_code)

# Results in:
# [<Response 302 at 0xb6a807cc>, <Response 302 at 0xb...
# 302
```

1.3.9 URL

Find the URL that the request was made to.:

```
async def main():
    r = await asks.get('http://example.com')
    print(r.url)

# Results in:
# 'http://example.com'
```

1.4 asks - Useful idioms and tricks.

1.4.1 Sanely making many requests (with semaphores)

A (bounded) semaphore is like a sofa (sofaphore?). It can only fit so many tasks at once. If we have a semaphore who's maximum size is `5` then only `5` tasks can sit on it. If one task finishes, another task can sit down. This is an extremely simple and effective way to manage the resources used by `asks` when making large amounts of requests.

If we wanted to request two thousand urls, we wouldn't want to spawn two thousand tasks and have them all fight for CPU time.

```
import asks
import curio

async def worker(sema, url):
    async with sema:
        r = await asks.get(url)
        print('got ', url)

async def main(url_list):
    sema = curio.BoundedSemaphore(value=2)  # Set sofa size.
    for url in url_list:
        await curio.spawn(worker(sema, url))

url_list = ['http://httpbin.org/delay/5',
            'http://httpbin.org/delay/1',
            'http://httpbin.org/delay/2']

curio.run(main(url_list))
```

This method of limiting works for the single-request `asks` functions and for any of the sessions' methods.

The result of running this is that the first and second url (`delay/5` and `delay/1`) run. `delay/1` finishes, and allows the third url, `delay/2` to run.

- After one second, `delay/1` finishes.
- After three seconds, `delay/2` finishes.
- After five seconds, `delay/5` finishes.

1.4.2 Maintaining Order

Due to the nature of async, if you feed a list of urls to `asks` in some fashion, and store the responses in a list, there is no guarantee the responses will be in the same order as the list of urls.

A handy way of dealing with this on an example `url_list` is to pass the enumerated list as a dict `dict(enumerate(url_list))` and then create a sorted list from a response dict. This sounds more confusing in writing than it is in code. Take a look:

```
import asks
import curio

results = {}

url_list = ['a', 'big', 'list', 'of', 'urls']
```

(continues on next page)

(continued from previous page)

```
async def worker(key, url):
    r = await s.get(url)
    results[key] = r

async def main(url_list):
    url_dict = dict(enumerate(url_list))
    for key, url in url_dict.items():
        await curio.spawn(worker(key, url))

sorted_results = [response for _, response in sorted(results.items())]

s = asks.Session(connections=10)
curio.run(main(url_list))
```

In the above example, `sorted_results` is a list of response objects in the same order as `url_list`.

There are of course many ways to achieve this, but the above is noob friendly. Another way of handling order would be a `heapq`, or managing it while iterating curio's taskgroups. Here's an example of that:

```
import asks
import curio

results = []
url_list = ["https://www.httpbin.org/get" for _ in range(50)]

s = asks.Session()

async def worker(key, url):
    r = await s.get(url)
    results.append((key, r.body))

async def main():
    async with curio.TaskGroup() as g:
        for key, url in enumerate(url_list):
            g.start_soon(worker, key, url)
        # Here we iterate the TaskGroup, getting results as they come.
        async for _ in g:
            print(f"done with {results[-1][0]}")

    sorted_results = [response for _, response in sorted(results)]
    print(sorted_results)
```

1.4.3 Handling response body content (downloads etc.)

The recommended way to handle this sort of thing is by streaming. The following examples use a context manager on the response body to ensure the underlying connection is always handled properly:

```
import asks
import curio

async def main():
    r = await asks.get('http://httpbin.org/image/png', stream=True)
    with open('our_image.png', 'ab') as out_file:
        async with r.body: # you can do the usual "as x" here if you like.
            async for bytechunk in r.body:
```

(continues on next page)

(continued from previous page)

```
                out_file.write(bytechunk)

curio.run(main())
```

An example of multiple downloads with streaming:

```
import asks
import curio

from functools import partial

async def downloader(filename, url):
    r = await asks.get(url, stream=True)
    async with curio.aopen(filename, 'ab') as out_file:
        async with r.body:
            async for bytechunk in r.body:
                out_file.write(bytechunk)

async def main():
    for indx, url in enumerate(['http://placehold.it/1000x1000',
                                'http://httpbin.org/image/png']):
        func = partial(downloader, str(indx) + '.png')
        await curio.spawn(func(url))

curio.run(main())
```

The `callback` argument lets you pass a function as a callback that will be run on each byte chunk of response body *as the request is being processed* . A simple use case for this is downloading a file.

Below you'll find an example of a single download of an image with a given filename, and multiple downloads with sequential numeric filenames. They are very similar to the streaming examples above.

We define a callback function `downloader` that takes bytes and saves 'em, and pass it in.

```
import asks
import curio

async def downloader(bytechunk):
    async with curio.aopen('our_image.png', 'ab') as out_file:
        await out_file.write(bytechunk)

async def main():
    r = await asks.get('http://httpbin.org/image/png', callback=downloader)

curio.run(main())
```

What about downloading a whole bunch of images, and naming them sequentially?

```
import asks
import curio

from functools import partial

async def downloader(filename, bytechunk):
    async with curio.aopen(filename, 'ab') as out_file:
        await out_file.write(bytechunk)
```

(continues on next page)

(continued from previous page)

```
async def main():
    for indx, url in enumerate(['http://placehold.it/1000x1000',
                                'http://httpbin.org/image/png']):
        func = partial(downloader, str(indx) + '.png')
        await curio.spawn(asks.get(url, callback=func))

curio.run(main())
```

1.4.4 Resending an `asks.Cookie`

Simply reference the `Cookie` 's `.name` and `.value` attributes as you pass them in to the `cookies` argument.

```
import asks
import curio

a_cookie = previous_response_object.cookies[0]

async def example():
    cookies_to_go = {a_cookie.name: a_cookie.value, 'another': 'cookie'}
    r = await asks.get('http://example.com', cookies=cookies_to_go)

curio.run(example())
```

CHAPTER 2

What is asks?

`asks` is an async HTTP lib that can best be described as an effort to bring the same level of usable abstraction that `requests` offers synchronous Python, to asynchronous Python programming. Ideal for API interactions, webscraping etc.

`asks` is compatible with `curio` and `trio`.

It is important to note that the code examples in this documentation are to showcase `asks`, and not `curio` or `trio`. In real code, it would be beneficial to use things like taskgroups/nurserys and other neat tools to manage your requests. Here's a link to `curio` and `trio`'s docs for reference:

http://curio.rtfd.io/

http://trio.rtfd.io

CHAPTER 3

Installation:

asks requires Python 3.6.2 or newer.

The easiest way to install `asks` is to pip it:

```
pip install asks
```

Internally `asks` uses the excellent h11. It will be installed automatically.

`asks` was built for use with curio and trio

CHAPTER 4

Importing `asks`

The following code will run the `example` coroutine once with `curio` and once with `trio`:

```
import asks
import curio
import trio

async def example():
    r = await asks.get('https://example.org')

curio.run(example)

trio.run(example)
```

CHAPTER 5

A quick note on the examples in these docs

`asks` began by only supporting `curio`, and the code examples use `curio` throughout. At any point in the examples you could switch say, `async with curio.TaskGroup` to `async with trio.open_nursery`, and everything would be the same bar `curio`/`trio`'s API differences. Internally, `asks` has no bias for either library. Both are beautiful creatures.

CHAPTER 6

A little example:

Here's how to grab a single request and print its content:

```
# single_get.py
import asks
import curio

async def grabber(url):
    r = await asks.get(url)
    print(r.content)

curio.run(grabber('https://example.com'))

# Results in:
# b'<!doctype html>\n<html>\n<head>\n    <title>Example Domain</title>\n\n
```

Making one request in an async program is a little weird, but not without its uses. This sort of basic `asks.get()` would slot in quite nicely in a greater program that makes some calls here and there.

CHAPTER 7

A bigger little example:

Here's an example of making 1000 calls to an API and storing the results in a list. We'll use the `Session` class here to take advantage of connection pooling.:

```
# many_get.py
# make a whole pile of api calls and store
# their response objects in a list.
# Using the homogeneous-session.

import asks
import curio

path_list = ['a', 'list', 'of', '1000', 'paths']

retrieved_responses = []

s = asks.Session('https://some-web-service.com',
                 connections=20)

async def grabber(a_path):
    r = await s.get(path=a_path)
    retrieved_responses.append(r)

async def main(path_list):
    for path in path_list:
        curio.spawn(grabber(path))

curio.run(main(path_list))
```

Now we're talkin'.

A thousand `requests` running async at the drop of a hat, using clean burning connection pooling to play nicely with the target server.

CHAPTER 8

Why asks?

If you like async, but don't like the spaghetti-docs future-laden many-looped `asyncio` lib, you'll probably love `curio` and `trio`. If you wish you could marry them with requests, you'll probably love `asks`.

Nice libs like `aiohttp` suffer the side effect of ugliness due to being specifically for `asyncio`. Inspired by `requests` and the fancy new-age async libs, I wanted to take that lovely ultra abstraction and apply it to an async HTTP lib to alleviate some of the pain in dealing with async HTTP.

CHAPTER 9

Features

`asks` packs most if not all of the features `requests` does. The usual `.json()`-ing of responses and such. You can take a more in-depth look here.

Because `asks` is aimed at crunching large piles of requests, its `Session` has some features you may not be aware of. Sessions in `asks` are the main focus. More detail can be found here

CHAPTER 10

The Future

Now that `trio` support has been implemented, it's housecleaning time. `asks` has some cobwebs that need clearing, and refactoring those in to a nice silk dress is the current focus.

CHAPTER 11

Contributing

Contributions are very welcome :)

CHAPTER 12

About

`asks` was created by Mark Jameson

https://theelous3.net

Shoutout to the fine folks of 8banana and co.