Librería requests, httpx

Antonio Espín Herranz

Librería requests

Librería requests

- Librería Python para peticiones HTTP.

- Objetivo
  - Realizar peticiones HTTP de una forma simple y sencilla.

- https://docs.python-requests.org/en/latest/

- Instalación (pypi)
  - https://pypi.org/project/requests/
  - **pip install requests**

- Github
  - https://github.com/psf/requests

Para ejecutar peticiones

- Podemos utilizar un servicio de request & response de la red:
- https://httpbin.org/

Peticiones con requests

- import request
- r = requests.get('https://api.github.com/events’)

- Petición con datos:
- r = requests.post('https://httpbin.org/post', data={'key': 'value'})

Peticiones: Verbos HTTP

- >>> r = requests.put('https://httpbin.org/put', data={'key': 'value'})
- >>> r = requests.delete('https://httpbin.org/delete')
- >>> r = requests.head('https://httpbin.org/get')
- >>> r = requests.options('https://httpbin.org/get')

Parámetros en la URL

- Estos datos se proporcionarían como pares clave / valor en la URL después de un signo de interrogación, por ejemplo httpbin.org/get?key=val.

- Requests le permite proporcionar estos argumentos como un diccionario de cadenas, utilizando el argumento **params**. Como ejemplo, si quisiera pasar key1=value1y key2=value2 para:
  - https://httpbin.org/get

- >>> payload = {'key1': 'value1', 'key2': 'value2'}
- >>> r = requests.get('https://httpbin.org/get', params=payload)

- Se traduce por: https://httpbin.org/get?key2=value2&key1=value1

Parámetros en la URL

- Se pueden pasar listas de parámetros:
- payload = {'key1': 'value1', 'key2': ['value2', 'value3']}

- >>> r = requests.get('https://httpbin.org/get', params=payload)
- >>> print(r.url)
- https://httpbin.org/get?key1=value1&key2=value2&key2=value3

La respuesta

- La petición get devuelve un objeto de tipo:
  - requests.models.**Response**

- Propiedades:
  - r.**text**: la respuesta del server en texto
  - r.**content**: en bytes, puede ser útil para imágenes.
  - r.**encoding**: La codificación, si cambiamos la codificación:
    - r.encoding =‘ISO-8859-1’ si se cambia requests utiliza la nueva codificación para cada petición nueva que se realice.

La respuesta

- El contenido de la respuesta también la podemos obtener en json:
  - >>> r = requests.get('https://api.github.com/events')
  - >>> r.json()

  - Se puede generar una excepción de tipo:
    - requests.exceptions.JSONDecodeError
    - Si la respuesta no tiene contenido (código 204)
    - O viene un json no válido
    - Por ejemplo: Una petición a: https://httpbin.org/

La respuesta sin procesar

- Se puede obtener la respuesta sin ningún tipo de procesamiento.
  - **r.raw**

  - >>> r = requests.get('https://api.github.com/events', **stream=True**)
  - >>> r.raw
  - <urllib3.response.HTTPResponse object at 0x101194810>

  - >>> r.raw.read(10)
  - '\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\x03'

La respuesta sin procesar

- Para grabar en un fichero se puede utilizar un iterador:

```
with open(filename, 'wb') as fd:
    for chunk in r.iter_content(chunk_size=128):
        fd.write(chunk)
```

- Cuando se transmite una descarga, lo anterior es la mejor forma y recomendada de recuperar el contenido.

- El tamaño de chunk_size se puede ajustar.

Personalizar encabezados

- >>> url = 'https://api.github.com/some/endpoint'
- >>> headers = {'user-agent': 'my-app/0.0.1'}
- >>> r = requests.get(url, headers=headers)

- Los encabezados personalizados tienen menos prioridad que las fuentes de información más específicas

Peticiones POST

- Envío de datos mediante una petición POST:
- Para enviar datos codificados en un formulario, o a un formulario HTML, utilizaremos el parámetro **data** con un diccionario.

- >>> payload = {'key1': 'value1', 'key2': 'value2'}

- >>> r = requests.post("https://httpbin.org/post", **data=payload**)
- >>> print(r.text)
- {
- ...
- "form": {
- "key2": "value2",
- "key1": "value1"
- },
- ...
- }

Peticiones POST

- Dentro del argumento data también nos podemos encontrar claves con mas de 1 valor, por ejemplo: tuplas o claves de diccionario con más de un valor.

- >>> **payload_tuples = [('key1', 'value1'), ('key1', 'value2')]**
- >>> r1 = requests.post('https://httpbin.org/post', data=payload_tuples)
- **>>> payload_dict = {'key1': ['value1', 'value2']}**
- >>> r2 = requests.post('https://httpbin.org/post', data=payload_dict)
- >>> print(r1.text)

```
{
 ...
 "form": {
   "key1": [
     "value1",
     "value2"
   ]
 },
 ...
}
```

```
>>> r1.text == r2.text
True
```

LISTAS DE TUPLAS O
DICCIONARIOS: CLAVE STRING, VALORES: LIST

Peticiones POST

- Se puede utilizar el módulo json para convertir objetos Python a Json (dump - graba en fichero, dumps - lo pasa a una cadena, load - carga un fichero, loads - de cadena a objetos Python).

- >>> import json

- >>> url = 'https://api.github.com/some/endpoint'
- >>> payload = {'some': 'data'}
- >>> r = requests.post(url, data=json.dumps(payload))

Publicar archivos

- Muy importante: Los archivos siembre abrirlos como binarios. Si se abren como texto, pueden dar problemas en el tamaño del archivo.

- >>> url = 'https://httpbin.org/post'
- >>> files = {'file': open('report.xls', 'rb')}

- >>> r = requests.post(url, files=files)
- >>> r.text
- {
- ...
- "files": {
- "file": "<censored...binary...data>"
- },
- ...
- }

Publicar archivos

- Al enviar los archivos se puede indicar los encabezados de forma específica:
- >>> url = 'https://httpbin.org/post'
- >>> files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel', {'Expires': '0'})}

- >>> r = requests.post(url, files=files)
- >>> r.text
- {
- ...
- "files": {
- "file": "<censored...binary...data>"
- },
- ...
- }

Publicar archivos

- Se pueden enviar cadenas como si fueran archivos.

- >>> url = 'https://httpbin.org/post'
- >>> files = {'file': ('report.csv', 'some,data,to,send\nanother,row,to,send\n')}

- >>> r = requests.post(url, files=files)
- >>> r.text
- {
- ...
- "files": {
- "file": "some,data,to,send\\nanother,row,to,send\\n"
- },
- ...
- }

Códigos de estado de respuesta

- Siempre que hagamos una petición es conveniente consultar el código de respuesta.
- >>> r = requests.get('https://httpbin.org/get')
- >>> r.status_code
- 200

- Dentro de la librería disponemos de la propiedad codes que vienen los código de respuesta codificados.
  - **r.status_code = requests.codes.ok**

Códigos de estado de respuesta

- Si hicimos una petición incorrecta por parte del cliente y obtenemos un error 4XX o un error en el servidor 5XX se puede generar con response.raise_for_status()

- >>> bad_r = requests.get('https://httpbin.org/status/404')
- >>> bad_r.status_code
- 404

- >>> bad_r.raise_for_status() → si el Código es 200 → obtenemos None
- Traceback (most recent call last):
- File "requests/models.py", line 832, in raise_for_status
- raise http_error
- requests.exceptions.HTTPError: 404 Client Error

Encabezados de la respuesta

- Los encabezados de la respuesta se pueden consultar a partir de la propiedad “headers” (es un diccionario de Python).
- >>> r.headers
- {
- 'content-encoding': 'gzip',
- 'transfer-encoding': 'chunked',
- 'connection': 'close',
- 'server': 'nginx/1.0.4',
- 'x-runtime': '148ms',
- 'etag': '"e1ca502697e5c9317743dc078f67693f"',
- 'content-type': 'application/json'
- }

Cookies

- Si la respuesta tiene cookies podemos acceder a ellas mediante la propiedad cookies:

- >>> url = 'http://example.com/some/cookie/setting/url'
- >>> r = requests.get(url)
- >>> r.**cookies**['example_cookie_name']
- 'example_cookie_value'

Cookies

- Envío de cookies al servidor:
- >>> url = 'https://httpbin.org/cookies'
- >>> cookies = dict(cookies_are='working')

- >>> r = requests.get(url, cookies=cookies)
- >>> r.text
- '{"cookies": {"cookies_are": "working"}}'

Cookies

- Las cookies se devuelven en un RequestsCookieJar, es un diccionario y proporciona una interface más completa para configurar las cookies a enviar (se envían con la propiedad **cookies**):

  - >>> jar = requests.cookies.**RequestsCookieJar**()
  - >>> jar.set('tasty_cookie', 'yum', domain='httpbin.org', path='/cookies')
  - >>> jar.set('gross_cookie', 'blech', domain='httpbin.org', path='/elsewhere')
  - >>> url = 'https://httpbin.org/cookies'
  - >>> r = requests.get(url, **cookies**=jar)
  - >>> r.text
  - '{"cookies": {"tasty_cookie": "yum"}}'

Redirección / Historial

- Las redirecciones se pueden rastrear mediante Response.history, contiene una lista con los objetos Response que se crearon.
- Esta lista se encontrará vacía en caso de no haber redirección.
- >>> r=requests.get('http://github.com/')
- >>> r.url
- 'https://github.com/'
- >>> r.status_code
- 200
- >>> r.history
- [<Response [301]>]

Redirección / Historial

- Con peticiones del tipo: GET, OPTIONS, POST, PUT, PATCH o DELETE
- Se puede deshabilitar el manejo de redirecciones con el parámetro **allow_redirect.**

- >>> r = requests.get('http://github.com/', **allow_redirects=False**)
- >>> r.status_code
- 301
- >>> r.history
- []

Redirección / Historial

- En caso de que la petición sea HEAD, se puede activar la redirección:

- >>> r = requests.head('http://github.com/', allow_redirects=True)
- >>> r.url
- 'https://github.com/'
- >>> r.history
- [<Response [301]>]

Timeout

- En un programa en producción siembre debería indicarse un timeout en la petición para evitar posibles bloqueos:

- >>> requests.get('https://github.com/', timeout=0.001)
- Traceback (most recent call last):
- File "<stdin>", line 1, in <module>
- requests.exceptions.Timeout: HTTPConnectionPool(host='github.com', port=80): Request timed out. (timeout=0.001)

Excepciones

- En el caso de un problema de red (por ejemplo, falla de DNS, conexión rechazada, etc.), las solicitudes generarán una excepción **ConnectionError**.

- Response.**raise_for_status**() generará un **HTTPError** si la solicitud HTTP devolvió un **código de estado incorrecto**.

- Si se agota el tiempo de espera de una solicitud (**Timeout**), se genera una excepción.

- Si una solicitud **excede el número configurado de redirecciones máximas**, se genera una excepción **TooManyRedirects**.

- Todas las **excepciones** que **Requests** genera explícitamente heredan **requests.exceptions.RequestException**.

Librería requests

Uso Avanzado

Objetos de sesión

- El objeto Session nos permite conservar información entre distintas solicitudes.
- Podemos almacenar cookies entre distintas sesiones.

- S = requests.Session()
- S.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
- r = s.get("http://httpbin.org/cookies")

- print(r.text)
- # '{"cookies": {"sessioncookie": "123456789"}}'

Objetos de sesión

- Se puede utilizar para proveer información por defecto a los métodos de peticiones.
- Se pueden asignar valores a un objeto de tipo Session:
  - S = requests.Session()
  - S.auth('user','pass')
  - S.headers.update({'x-test':'true'})

- # Se envían ambos datos x-test y x-test2
  - S.get('http://httpbin.org/headers', headers={'x-test2':'true'})

- Si pasamos un diccionario a la petición, será unido con los valores que fueron asignados a nivel de sesión.
  - Si alguna clave coincide se sobrescribe.

Objetos de sesión

- Para omitir valores asignados a nivel de sesión, se envía la clave con el valor None.

Objetos Request y Response

- Cada vez que se hace una **petición a un servidor** con alguno de los métodos: get, post, etc. Se está construyendo un objeto **Request**.
  - El objeto Request se envía al servidor para obtener información y después
  - Se construye un objeto **Response** con la **respuesta** del Servidor.

  - R = requests.get(url)
  - R.headers → las cabeceras que envío el server
  - R.request.headers → las cabeceras que se enviaron al server.

Validación de Certificados SSL

- requests puede verificar **certificados SSL** para peticiones HTTPS como un navegador Web. Utilizar el parámetro **verify**.
- Ejemplo:
  - requests.get('https://www.dpii.es', verify=True)
  - <Response [200]>

Workflow del cuerpo del contenido

- Cuando se realiza una petición el cuerpo de la respuesta se descarga automáticamente.

- Este comportamiento se puede cambiar hasta que se accede al contenido mediante: **Response.content** con el parámetro **stream=True**. Es como una descarga Lazy (demorada).

- En este caso sólo se han descargado las cabeceras, de tal forma que podemos controlar el tamaño del contenido y no descargar en caso de que sea demasiado grande.

Ejemplo

- mi_url = ‘…’
- R = requests.get(mi_url, **stream=True**)

- if int(r.headers[‘content-length’]) < MAX_SIZE:
  - contenido = R.content
- else:
  - Print(‘Demasiados datos’)

Workflow del cuerpo del contenido

- El workflow se puede controlar mediante los iteradores:
  - Response.**iter_content** y Response.**iter_lines**. Los dos métodos devuelven un generador.

  - OJO, puede ser ineficiente: si configuramos la petición como stream = True **requests no puede liberar la conexión** hasta que se consuman todos los datos o hagamos **Response.close**

Keep Alive

- urllib3, keep-alive es 100% automático dentro de una sesión! Cualquier petición que se ejecute dentro de una sesión, reutilizará la conexión apropiada!

Subir por streaming

- Requests permite subir ficheros por streaming. Se pueden enviar ficheros pesados sin tener que leerlos en memoria.

- with open(‘path_file’, ‘rb’) as f:
  - requests.post(‘url’, data=f)

Peticiones fragmentadas - Chunk-Encoded

- Se pueden utilizar generadores para enviar datos:

- def generador():
  - # No admite cadenas, tienen que ir como bytes
  - yield b’hola que tal’

- r = requests.get('https://httpbin.org',data=generador())

Peticiones en streaming

- Se pueden hacer peticiones en streaming con:
  - requests.Response.**iter_lines**()
  - OJO hay que configurar el parámetro **stream** a **True**.
  - Dentro de httpbin.org tenemos una opción para hacer peticiones basadas en streaming.

```
import json
import requests

r = requests.get('http://httpbin.org/stream/20', stream=True)

for line in r.iter_lines():

    # filter out keep-alive new lines
    if line:
        print json.loads(line)
```

Proxies

- Si la conexión hay que realizarla a través de un proxy se pueden configurar con un diccionario y utilizar el parámetro: **proxies** de la petición.

- import requests
- proxies = {
- "http": "http://10.10.1.10:3128",
- "https": "http://10.10.1.10:1080",
- }

- requests.get("http://example.org", proxies=proxies)

Proxies

- O a través de variables de entorno:
- $ export HTTP_PROXY="http://10.10.1.10:3128"
- $ export HTTPS_PROXY="http://10.10.1.10:1080"
- $ python3

- >>> import requests
- >>> requests.get("http://example.org")

Verbos HTTP

- Requests provee acceso a casi todo el rango de verbos HTTP: GET, OPTIONS, HEAD, POST, PUT, PATCH y DELETE

- HTTP GET es un método idempotente el cual regresa un recurso a partir de una URL; por lo tanto, este verbo es utilizado cuando se quiere obtener información desde una ubicación web.

- import requests
- >>> r = requests.get(‘url')

- >>> if (r.status_code == requests.codes.ok):
- ... print r.headers['content-type']

Verbos HTTP

- El verbo OPTIONS soportado por Requests para ver que tipos de métodos HTTP están soportados.

- Si el proveedor no implementa el método options, obtendremos un error:
  - >>> verbs = requests.options(r.url)
  - >>> verbs.status_code
  - 500

- Si va todo bien:
  - >>> verbs = requests.options('http://a-good-website.com/api/cats')
  - >>> print verbs.headers['allow']
  - GET,HEAD,POST,OPTIONS

Link Headers

- Muchas APIs soportan Link headers. Estas cabeceras hacen que las APIs sean más auto-descriptivas y detectables.

- >>> url = 'https://api.github.com/users/kennethreitz/repos?page=1&per_page=10'
- >>> r = requests.head(url=url)
- >>> r.headers['link']
- '<https://api.github.com/users/kennethreitz/repos?page=2&per_page=10>; rel="next", <https://api.github.com/users/kennethreitz/repos?page=6&per_page=10>; rel="last"'

Timeouts

- La mayoría de las peticiones externas deben tener un timeout anexo, en caso de que el servidor no esté respondiendo a tiempo.

- El timeout connect es el número de segundos que Request esperará para que tu cliente establezca una conexión a una máquina remota (correspondiente al método connect()) en el socket. Es una buena práctica establecer tiempos de conexión a algo un poco más grande que un múltiplo de 3, para permitir el tiempo por defecto TCP retransmission window.

- Una vez que tu cliente se ha conectado al servidor y enviado la petición HTTP, el timeout read es el número de segundos que el cliente esperará para que el servidor envie una respuesta.

Timeouts

- Si especificas un solo valor para el timeout, como esto:
  - r = requests.get('https://github.com', timeout=5)

- El valor de timeout será aplicado a ambos timeouts: connect y read. Especifique una tupla si deseas establecer el valor separadamente:
  - r = requests.get('https://github.com', timeout=(3.05, 27))

- Si el servidor remoto es demasiado lento, puedes decirle a Request que espere por siempre la respuesta, pasando None como el valor de timeout.
  - r = requests.get('https://github.com', timeout=None)

API de referencia

- https://docs.python-requests.org/en/latest/api/

Librería httpx

Librería httpx

- Instalación: https://pypi.org/project/httpx/
- **pip install httpx**
- https://www.python-httpx.org/

- Es un cliente HTTP para Python 3, proporciona APIs para generar peticiones Web síncronas y asíncronas.

- Realiza peticiones a una web y obtenemos toda la información del sitio en forma de propiedades: contenidos, tipo de contenido, cookies, etc.

Métodos HTTP

- Implementa los métodos del protocolo HTTP para interactuar con una Web:
  - **GET**
    - Para solicitar un recurso. Muy fácil para extraer información. Se pueden enviar parámetros
  - **POST**
    - Envío de una entidad a un recurso específico. Normalmente provoca un cambio de estado en el servidor.
  - **DELETE**
    - Borrar un recurso específico del servidor
  - **PUT**
    - Para reemplazar un recurso en el servidor.

METODOS HTTP

- **HEAD**
  - Pide una respuesta idéntica a la petición GET, pero sin el cuerpo de la respuesta.
- **OPTIONS**
  - Utilizado para describir las opciones de comunicación para el recurso destino.
- **PATCH**
  - Se utiliza para aplicar modificaciones parciales a un recurso.
- https://developer.mozilla.org/es/docs/Web/HTTP/Methods

Funciones httpx

- **httpx.get(“url”)**
  - Se puede recoger la respuesta en un objeto, si no indicará el código de respuesta. 200 → ok!
  - El objeto que devuelve es: httpx.Response
  - Extraer información del sitio.

  - Con parámetros:
    - Se pasan mediante un diccionario (clave valor, parámetros de la querystring)
    - httpx.get(“URL”, **params**={“key”:”value”})

Funciones httpx

- **httpx.get(“url”)**
  - Con **headers** personalizadas:
    - url = 'https://httpbin.org/headers’
    - headers = {'user-agent': 'my-app/0.0.1’}
    - r = httpx.get(url, **headers**=headers)

- Recuperación de datos binarios:
  - r = httpx.get(“URL_IMAGEN”)

  - from PIL import Image
  - from io import BytesIO
  - i = Image.open(BytesIO(r.content))
  - i.show() # Para mostrarla

Funciones httpx

- httpx.**post**(“URL”, **data**=data)
  - Envío de peticiones POST con datos codificados en un diccionario.
  - data = {“key1”:”Val1”, “key2”:”Val2”}

  - Se puede codificar con valores múltiples:
    - data = {'key1': ['value1', 'value2']}

Funciones httpx

- httpx.**post**(“URL”, **files**=files)
  - files = {'upload-file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel')}
  - r = httpx.post("https://httpbin.org/post", files=files)
  - print(r.text)

Funciones httpx

- **httpx.stream(“método”,”url”)**
  - Similar a la anterior pero no carga todo en memoria. Utilizamos un generador.
  - g = httpx.stream(“GET”, “URL)
  - for i in g.gen:
    - print(i)

  - Propiedades del objeto devuelto: args, func, gen, kwds

Propiedades httpx

- **httpx.codes**
  - Enumeración con todos los códigos de respuesta del protocolo http.
  - Imprimir los códigos.
  - Significado de los códigos: https://developer.mozilla.org/es/docs/Web/HTTP/Status

Funciones de ayuda

- Disponemos del método **request** para hacer todo tipo de peticiones.
- httpx.**request**(method, url, *, params=None, content=None, data=None, files=None, json=None, headers=None, cookies=None, auth=None, proxies=None, timeout=Timeout(timeout=5.0), follow_redirects=False, verify=True, cert=None, trust_env=True)
  - Por el parámetro method: get, post, put, etc.
  - El resto de parámetros son de configuración de la petición:
    - Data (para los parámetros)
    - Files (ficheros para subir)
    - Headers (envío cabeceras)
    - Verify: Para verificación de https.
    - Etc.

Funciones de ayuda

```
>>> import httpx
>>> response = httpx.request('GET', 'https://httpbin.org/get')
>>> response
<Response [200 OK]>
```

- El método request nos sirve para hacer cualquier tipo de petición.

Funciones de ayuda

- httpx.get(url , * , params = None , headers = None , cookies = None , auth = None , proxies = None , follow_redirects = False , cert = None , verify = True , timeout = Timeout (tiempo de espera = 5.0) , trust_env = True)
- Envía una solicitud GET.

- Parámetros : ver httpx.request.

- Los parámetros data, filesy json no están disponibles en esta función, ya que las soliciudes GET no deben incluir un cuerpo de la petición.

Funciones de ayuda

- De forma similar a get tenemos:
  - options
  - head
  - post
  - put
  - patch
  - delete

Gestión de Excepciones

- Las mas importantes clases de excepción en HTTPX son RequestError y HTTPStatusError:

```
try:
    response = httpx.get("https://www.example.com/")
except httpx.RequestError as exc:
    print(f"An error occurred while requesting {exc.request.url!r}.")

response = httpx.get("https://www.example.com/")
try:
    response.raise_for_status()
except httpx.HTTPStatusError as exc:
    print(f"Error response {exc.response.status_code} while requesting
    {exc.request.url!r}.")
```

Gestión de Excepciones

- También hay una clase base HTTPError que incluye ambas categorías y se puede utilizar para detectar solicitudes fallidas o respuestas **4xx y 5xx.** Recursos no encontrados, errores en el servidor …

- Puede usar esta clase base para capturar ambas categorías ...

```
try:
    response = httpx.get("https://www.example.com/")
    response.raise_for_status()
except httpx.HTTPError as exc:
    print(f"Error while requesting {exc.request.url!r}.")
```

Gestión de Excepciones

- O maneje cada caso explícitamente:
- try:
  - response = httpx.get("https://www.example.com/")
    response.raise_for_status()
- except httpx.RequestError as exc:
  - print(f"An error occurred while requesting {exc.request.url!r}.")
- except httpx.HTTPStatusError as exc:
  - print(f"Error response {exc.response.status_code} while requesting {exc.request.url!r}.")

Objeto Client

- class httpx.**Client**(*, auth=None, params=None, headers=None, cookies=None, verify=True, cert=None, http1=True, http2=False, proxies=None, mounts=None, timeout=Timeout(timeout=5.0), follow_redirects=False, limits=Limits(max_connections=100, max_keepalive_connections=20, keepalive_expiry=5.0), max_redirects=20, event_hooks=None, base_url='', transport=None, app=None, trust_env=True)

```
>>> client = httpx.Client()
>>> response = client.get('https://example.org')
```

Objeto Client: Parámetros I

- auth : (opcional) una clase de autenticación para usar al enviar solicitudes.
- params : (opcional) parámetros de consulta para incluir en las URL de solicitud, como una cadena, diccionario o secuencia de dos tuplas.
- headers : (opcional) Diccionario de encabezados HTTP para incluir al enviar solicitudes.
- cookies : (opcional) Diccionario de elementos de cookies para incluir al enviar solicitudes.
- verify : (opcional) Certificados SSL (también conocidos como paquete CA) que se utilizan para verificar la identidad de los hosts solicitados. O True(paquete de CA predeterminado), una ruta a un archivo de certificado SSL, o False (que deshabilitará la verificación).
- cert : (opcional) un certificado SSL utilizado por el host solicitado para autenticar al cliente. Ya sea una ruta a un archivo de certificado SSL, o dos tuplas de (archivo de certificado, archivo de clave) o tres tuplas de (archivo de certificado, archivo de clave, contraseña).

Objeto Client: Parámetros II

- proxies : (opcional) un diccionario que asigna claves de proxy a URL de proxy.
- timeout : (opcional) la configuración de tiempo de espera que se utilizará al enviar solicitudes.
- limits : (opcional) la configuración de límites que se utilizará.
- max_redirects : (opcional) el número máximo de respuestas de redireccionamiento que se deben seguir.
- base_url : (opcional) una URL que se utilizará como base al crear las URL de solicitud.
- transport : (opcional) una clase de transporte que se utiliza para enviar solicitudes a través de la red.
- app : (opcional) una aplicación WSGI a la que enviar solicitudes, en lugar de enviar solicitudes de red reales.
- trust_env : (opcional) habilita o deshabilita el uso de variables de entorno para la configuración.

Objeto AsyncClient

```
import httpx
import asyncio

async def main():
    async with httpx.AsyncClient() as client:
        r = await client.get('http://test.webcode.me')
        print(r.text)

asyncio.run(main())
```

Objeto Request

- Permite configurar los parámetros de una petición para luego enviarla:

- Ejemplo:
  - >>> client = Client()
  - >>> request = httpx.Request("GET", "https://example.org", headers={'host': 'example.org’})
  - >>> response = client.send(request)

Objeto: httpx.Response

- Representa la respuesta de una petición.
  - **Propiedades**:
    - .status_code int
    - .url URL
    - .headers Headers
    - .text str
    - .encoding str
    - .cookies Cookies
    - .history List[Response]
    - .is_redirect bool
    - Content bytes

- https://www.python-httpx.org/api/#response

Objeto httpx.Response

- **Métodos**:
  - def json(): Podemos obtener la respuesta en JSON:
    - r = httpx.get('https://api.github.com/events')
    - r.json()
  - def .raise_for_status() -> None
    - Lanza excepciones para respuestas 4xx o 5xx
  - def .read() -> bytes
  - Iteradores:
    - def .iter_raw() - bytes iterator
    - def .iter_bytes() - bytes iterator
    - def .iter_text() - text iterator
    - def .iter_lines() - text iterator
  - def close():
    - Para cerrar la respuesta y libera la conexión

Objeto httpx.Response

- Métodos: (método asíncronos)
  - def .aread() - bytes
  - def .aiter_raw() - async bytes iterator
  - def .aiter_bytes() - async bytes iterator
  - def .aiter_text() - async text iterator
  - def .aiter_lines() - async text iterator
  - def .aclose() - None
    - Cierra la respuesta y libera la conexión. Se llama automáticamente si el cuerpo de la respuesta se lee hasta el final.
  - def .anext() - Response

Objeto URL

- Podemos extraer cierta información de la url, se crea a partir de la Url.
- def __init__(url, allow_relative=False, params=None)
- .scheme - str
- .authority - str
- .host - str
- .port - int
- .path - str
- .query - str
- .raw_path - str
- .fragment - str
- .is_ssl - bool
- .is_absolute_url - bool
- .is_relative_url - bool

```
url = URL(http://www.elpais.es)
print(url.propiedad)
```

Headers

- Es un multi-dict:
  - >>> headers = Headers({'Content-Type': 'application/json’})
  - >>> headers['content-type’]
  - 'application/json'

Cookies

- >>> cookies = Cookies()
- >>> cookies.set("name", "value", domain="example.org")

- def __init__(cookies: [dict, Cookies, CookieJar])
- .jar- CookieJar
- def extract_cookies(response)
- def set_cookie_header(request)
- def set(name, value, [domain], [path])
- def get(name, [domain], [path])
- def delete(name, [domain], [path])
- def clear([domain], [path])

httpx vs request

- Podemos ver una comparativa de las dos librerías en el siguiente enlace:
  - https://www.confessionsofadataguy.com/httpx-vs-requests-in-python-performance-and-other-musings/