# Capitolo 1

## Python requests

**Installazione** `pip install requests` `from requests import *` `pip install beautifulsoup4` `from bs4 import BeautifulSoup`

### Metodi

Nella libreria requests, ogni metodo HTTP corrisponde a una funzione.

Chiamando la funzione di un metodo, ad esempio `get`: `response = get('https://api.github.com')` ci viene restituito un oggetto `Response` che contiene molte informazioni sulla risposta che ci è stata restituita, tra cui: - ##### Status code: `response.status_code` Notare che `response.status_code` è un intero, mentre `response` risulta `True` se lo status code è compreso tra 200 e 400, `False` altrimenti. (Se vuoi capire come questo sia possibile, puoi dare un'occhiata al method overloading) - ##### Contenuto: `response.text` In questo modo possiamo vedere cosa ci è stato restituito dal server, cosa avremmo visto se avessimo visitato lo stesso link da un browser. `response.content` fa la stessa cosa, ma restituisce bytes invece che una stringa. - ##### Contenuto in JSON: `response.json()` Particolarmente utile quando abbiamo a che fare con delle API. Otterremmo lo stesso risultato usando `.text` e deserializzando il risultato con `json.loads(response)` - ##### Headers: `response.headers` Che restituisce un oggetto simile a un dizionario ma con key case-insensitive. Quindi se vogliamo accedere ad un header in particolare, possiamo specificarlo: `response.headers['content-type']`

### Personalizzazione della richiesta

Come visto nel capitolo precedente, ci sono diversi tipi di scambio di informazioni che permettono di personalizzare una richiesta:

```
response = get(
    'https://it.wikipedia.org/w/index.php',
    params={'search': 'capture+the+flag'},
)
```

```
response = get(
    'https://it.wikipedia.org/w/index.php',
    params={'search': 'capture+the+flag'},
    headers={'User-Agent': 'Mozilla/5.0'},
)
```

### Altri metodi

```
post('https://httpbin.org/post', data={'key':'value'})
put('https://httpbin.org/put', data={'key':'value'})
delete('https://httpbin.org/delete')
head('https://httpbin.org/get')
patch('https://httpbin.org/patch', data={'key':'value'})
options('https://httpbin.org/get')
```

### Sessioni

In caso fosse necessario eseguire più azione tramite una sola connessione (esempio: passiamo per diverse API che ci assegnano e controllano cookie/header), è possibile usare l'oggetto `Session`.

```
s = Session()

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

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

Tra i tanti possibili casi d'uso, le sessioni possono risultare particolarmente utili nelle attacco e difesa che propongono servizi nei quali bisogna registrarsi/loggarsi per ottenere la flag. In questi casi, può risultare comodo usare le sessioni sfruttando i `context manager`:

```
with requests.Session() as session:
    session.auth = ('randomuser', 'randompass')

    session.post('https://api.cyberchallenge.it/pwnedwebsite/register')
    session.post('https://api.cyberchallenge.it/pwnedwebsite/login')
    response =
        session.get('https://api.cyberchallenge.it/pwnedwebsite/idor/flag')
```

### Cookies

Come detto, in caso fossero coinvolti dei cookie nel processo da automatizzare, è il caso di utilizzare le sessioni in modo da non dover fare alcun intervento manuale.

In caso volessimo vedere o aggiungere dei cookie, basta sapere che essi sono salvati in un dizionario, quindi per ottenerli basterà un `session.cookies.get_dict()`

Per una visualizzazione "pulita" dei vari parametri del cookie (grazie Bobby)

```
import requests

response = requests.get('http://google.com', timeout=30)

# {'AEC':
    'Ad49MVE4KO7sQX_pRIifPtDvL666jJcj34BmOFeETG9YU_1mu1SINQN-Q_A'}
print(response.cookies.get_dict())

result = [
    {'name': c.name, 'value': c.value, 'domain': c.domain,
        'path': c.path}
    for c in response.cookies
]

# [{'name': 'AEC', 'value':
    'Ad49MVGjcnQKK55wgCKVdZpw4PDgEgicIVB278lObJdf4eXaYChtDZcGLA',
    'domain': '.google.com', 'path': '/'}]
print(result)
```

**Aggiungere un cookie alla sessione** La libreria requests usa i CookieJar per gestire i cookie. Per aggiungere un cookie alla CookieJar della sessione, si può usare il metodo `update`:

```
from requests import *
s = Session()
s.cookies.update({'username': 'Francesco Titto'})
response = s.get('http://ctf.cyberbootcamp.it:5077/')
```

In particolare, i metodi `sessione.cookie.XYZ` aiutano ad interfacciarsi con i CookieJar. Esistono molti metodi utili, ma ciò che è stato fino ad ora basta per quanto concerne lo scopo di questa guida.

**Tips&Tricks**

**Controllo dei metodi "permessi"** Come visto nello scorso capitolo, il metodo `OPTIONS` permette di visualizzare i metodi disponibili. Per fare questo, dopo aver eseguito una richiesta `OPTIONS`, il risultato desiderato sarà restituito nell'header `Allow`: `response.headers['allow']`

**Utilizzo dei giusti parametri** Abbiamo visto i diversi modi per mandare dei dati al server. È importante non fare confusione tra `params`, che manda parametri della query, `data`, che manda informazioni nel corpo della richiesta

(request body), e `json` che fa la stessa cosa convertendo in json il dizionario che gli diamo e settando l'header `Content-Type` ad `application/json`. Notare che inserire del json nel parametro `json`, esempio: `json=json.dumps(data)` risulterà in un doppio dump (e quindi vari errori di difficile comprensione).

**`robots.txt` e `sitemap.xml`** Può succedere in alcune challenge blackbox di CyberChallenge (ma soprattutto OliCyber) che alcune informazioni necessarie alla risoluzione della challenge (anche source code) siano indicati nel robots.txt o nella sitemap. Controllare non vi costa niente, e vi può far risparmiare molto tempo. È invece molto più raro in altre CTF (non mi è mai successo di trovarci qualcosa)

**Timeout** Per evitare che il programma si blocchi per una richiesta sbagliata o un problema infrastrutturale, è stato introdotto il `Timeout`: `get('https://api.github.com', timeout=1.5)`. È possibile inserire il numero di secondi (int o float) da lasciar passare prima che un errore venga triggerato. Se combinato col `Try/Except` può risultare utile per attacchi time-based (crittografia, sql ed altro).

## DOM

Premendo F12 nei principali browser, vengono aperti gli strumenti per sviluppatori. La prima sezione mostrata di default è `elements`, elementi, che ci permette di esplorare interattivamente il Document Object Model (DOM).

Il DOM è una struttura multi-piattaforma e indipendente dal linguaggio, tuttavia nel nostro caso la seguente definizione è sufficiente: il DOM è un'interfaccia che tratta HTML come una struttura ad albero dove ogni nodo è un oggetto che rappresenta parte del documento.

Se non si è mai avuto a che fare con HTML e/o il concetto di DOM, il modo migliore per capire come funziona e prenderci dimestichezza è proprio visitando siti che si conoscono bene (ad esempio, un articolo su wikipedia) ed usando la sezione `elements` degli strumenti per sviluppatori


Figure 1: Esempio di utilizzo degli strumenti per sviluppatori

Passando il cursore su uno degli elementi, questo verrà evidenziato.

Nell'esempio mostrato, `h3` è il tag dell'elemento, `post-title` la classe. Può essere presente anche l'`id`, che identifica univocamente l'elemento.

### BeautifulSoup

BeautifulSoup è una libreria estremamente utile per il web scraping. Si utilizza insieme alla libreria `requests` per ottenere automaticamente una serie di dati di nostro interesse.

```
import requests
from bs4 import BeautifulSoup

URL = "https://theromanxpl0.it/"
page = requests.get(URL)

soup = BeautifulSoup(page.content, "html.parser")

print(soup.prettify())
```

```
results = soup.find(id="penguin-login writeup")
```

```
results = soup.find_all("h3", class_="post-title")
resText = soup.find_all("h3", string="penguin")

for result in results:
    print(result.prettify(), end="\n")

for result in resText:
    print(result.prettify(), end="\n")
```

```
print(result.text, end="\n")
```

Per la struttura del DOM, esso ha una gerarchia, ovvero i contenuti sono uno dentro l'altro (quelli che vediamo sono tutti figli dell'elemento con tag `HTML`).

```
result = soup.find("h3", class_="post-title")
result = result.parent
print(result.text, end="\n")
```

**Estrarre i link** Gli elementi `a`, approssimando, rappresentano un link, che si trova come attributo `href`.

```
import requests
from bs4 import BeautifulSoup

URL = "https://theromanxpl0.it/"
page = requests.get(URL)

soup = BeautifulSoup(page.content, "html.parser")

links = soup.find_all("a")
for link in links:
    link_url = link["href"]
    print(f"writeup link: {link_url}\n")
```

*Esercizi: prime 16 a partire da questa: https://ctf.cyberchallenge.it/challenges#challenge-255 In caso non si avesse accesso alla piattaforma CyberChallenge, c'è un'alternativa pubblica qui: https://training.olicyber.it/challenges#challenge-340*

*L'introduzione è molto stringata e più orientata agli esempi in quanto l'argomento può diventare molto grande a seconda di quanto lo si vuole approfondire, e non mi aspetto che dobbiate usare questa libreria molto spesso, ancor meno se si tratta di un utilizzo non superficiale.*