

Practical Data Science & Engineering Vol.1

次目

1. 現代の最強の科学・意思決定法、定量分析

- 1.1 なぜ、定量分析が最強なのか

2. データを集める手段、スクレイピングの基礎

- 2.1 Requests + BeautifulSoupでスクレイピングする
- 2.2 Google Chrome + Seleniumでスクレイピングする
- 2.2.1 Pixivを例に取る
- 2.2.2 Pixivを例に取る: Google Chrome + Seleniumでデータを取得する
- 2.2.3 Pixivを例に取る: 自動でログインしてデータを取得する
- 2.3 ハイパーリンクが作るネットワークは探索問題
- 2.3.1 幅優先探索・深さ優先探索・ビームサーチ
- 2.4 法的問題

3. 最強のDB、自作DBを作る

- 3.1 自作KVSとその関連
- 3.2 最強のDBは結局ファイルシステム

4 . Depth 1, UserAgent, Referrer を偽装する

- 4.1 自分が使っているUserAgentを確認する
- 4.2 サイト管理人格に配慮する
- 4.3 Referrerを偽装する

5. Depth 2, IPを偽装する

- 5.1 プロキシサーバを立ててアクセス
- 5.1.2 閑話休題、AWSアカウントをバンされる
- 5.2 公開プロキシ経由でのアクセス
- 5.3 tor経由でのアクセス
- 5.3.1 Dockerでtorのsocks5 proxyサーバを立てる
- 5.3.2 環境変数でsocks5を指定する
- 5.3.3 pythonでsocks5を指定する

6. Depth 3, MultiCore, Multi Machineでスクレイピングする

- 6.1 Thread vs Multiprocessing
- 6.2 MultiprocessingをMulti Machineに拡張

7. フェアネスを考慮したスクレイピング

- 7.1 ランダムドメイン選択
- 7.2 スクレイピングの頻度を確率的にして調整

8. 練習

8.1 Practice 1, 無料のphotstockをスクレイピングして大量のフリー画像を集める

- 8.1.1 <https://unsplash.com/>
- 8.1.2 シンプルな全探索アルゴリズム（順序なし）

8.2 Practice 2, Bingの検索機能を利用して、大量のグラビア写真を集める

- TODO

8.3 Practice 3, YouTubeの動画をダウンロードする

- 8.3.1 YouTubeの動画のダウンロードに便利な [youtube-dl](#)
- 8.3.2 IPを使い潰すしかなさそう

9. 閑話休題1, GCPで間違ったクエリを送って事故ったときの話

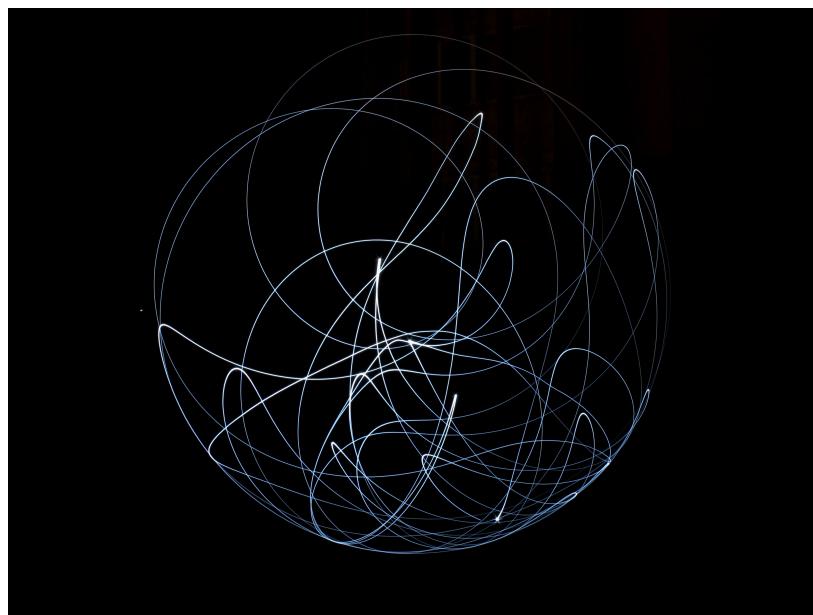
- TODO

10. 閑話休題2, リクエストが多すぎるとDNSが応答しなくなる

1. 現代の最強の科学・意思決定法、定量分析

1.1 なぜ、定量分析が最強なのか

エモい話をすると、世の中には簡単な法則で成り立っている事象と、複雑でカオス的な振る舞いをしてる事象の2つに大きく二分できます。簡単な法則とは、ニュートンの物理法則が適応できシンプルな演繹的な理論の導出が成立する世界になります。複雑でカオス的な振る舞いとは、木星の渦の模様や、何箇所の折り曲がるパーティで連結された振り子の振る舞いなどは鋭敏に初期値に依存し、解析的にはほとんど解くことが不可能になります。このカオス的な振る舞いは簡単な法則性が背景にあるだろと仮定して、法則性を見つけ出そうとすると、思わぬお落とし穴にハマることがあります。代表的な例としてエセ科学や反ワクチンなどシンプルな法則性が後ろにあるような仮定をおいてしまう例などがあります。



図x. 二重振り子の軌跡、この例ですら解析的に解くのは手計算では難しい

では、ワクチンの副反応が実際に存在するとか、その影響度合いなどは結局複雑だからわからない、という疑問が出るかと思います。この課題に対して、答えを与えるのが、定量分析になります。

ワクチンの治療において2群にうまく恣意性がなく分けることができ、追跡調査をできたとして以下の結果が得られたとします。（データはフィクションです）

表 x.

ワクチンを投与したグループ	ワクチンを投与しなかったグループ	
自己免疫疾患診断あり	31	62
自己免疫疾患診断なし	100023	200035

このデータが仮に得られたとしたら、どう判断すればよいでしょうか？

ほとんど発生確率に差がないことからこのワクチンでの副反応自体はあまり高くないことが伺えます。実際には、事象にはガウシアンノイズなどのランダムネスがあり、こんなきれいな結果にはなりませんが、サンプルしたサイズに対して確率的に許容できる幅があり、その範囲を逸脱しなければ、差がないと言えます。

この結果を通して統計学が大切だという点も言えますが、もっともこの著書で言いたいのは、定量化、つまりデータを大量に束ねて大きな数にすることである程度の真実が初めて見えていく、ということになります。Twitterなどでは、反ワクチンと呼ばれる方々が息子・娘や親戚や友達が副反応で今も苦しんでいる、という一部の証言から一般化した意見をそのまま鵜呑みにすればいいわではないわけです。

人間の本能により、原因と結果には単純でわかりやすいロジックを当てはめそうになります。それというのも、もう一つの人間の本能に腹落ちすること、納得することが大切であるというもう一つの本能があり、これにより事実の誤謬が世の中にはびこる結果になっています。

現代はインターネットにより誰もが手軽に様々な情報を手に入れる事が可能になりました。専門のサイト、サービスやSNSなどに多くの人が様々なことを報告しています。これらの情報を束ねると手軽に真実がわかることがあります。正しい情報とはそれだけで皆さんの判断にプラスの影響をもたらしますし、世界を見る視野を正しく広げる一助にもなります。そんなわけで、“Practical Data Science and Data Engineering Vol.1”ではスクレイピングのテクニックについてお伝えしていこうと思います。

2. データを集める手段、スクレイピングの基礎

2.1 Requests + BeautifulSoupでスクレイピングする

スクレイピングの手法としてPythonで一般的である、requestsとBeautifulSoupのモジュールを利用したスクレイピングについて説明を行います。環境はUbuntu LinuxかMacOSを前提とします。Windowsをお使いの方はAWSかGCPで安いインスタンスを借りることができますので、Linuxをインストールして体験してみると良いと思います。

requestsはpythonで扱いにくかったhttp, httpsなどのアクセスを簡略化して様々なベストプラクティスを詰め込んだライブラリです。他にも様々なものがありますが、これが2020年現在、もっとも使いやすいものです。

BeautifulSoupとはhtmlパーサライブラリで、htmlは特定のフォーマットで記述された言語になり、機械で適切に処理させるにはパーサというものを介さないといけません。

pipでのrequests, BeautifulSoupのインストール

Anacondaや特殊なPythonでは別のパッケージマネージャがありますが、pipで統一して話をす進めます。

```
$ pip install requests bs4
```

```
gimpei@Akagi:GIJYUTUSHOTEN8 % pip install requests bs4
Collecting requests
  Using cached https://files.pythonhosted.org/packages/51/bd/23c926cd341ea6b7dd0b2a00aba99ae0f828be89d72
b2190f27c11d4b7fb/requests-2.22.0-py2.py3-none-any.whl
Collecting bs4
  Using cached https://files.pythonhosted.org/packages/10/ed/7e8b97591f6f456174139ec089c769f89a94a1a4025
fe967691de971f314/bs4-0.0.1.tar.gz
Requirement already satisfied: idna<2.9,>=2.5 in /home/gimpei/.pyenv/versions/3.7.4/lib/python3.7/site-p
ackages (from requests) (2.8)
Requirement already satisfied: urllib3!=1.25.0,!<1.25.1,<1.26,>=1.21.1 in /home/gimpei/.pyenv/versions/3
.7.4/lib/python3.7/site-packages (from requests) (1.25.7)
Requirement already satisfied: certifi>=2017.4.17 in /home/gimpei/.pyenv/versions/3.7.4/lib/python3.7/si
te-packages (from requests) (2019.11.28)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /home/gimpei/.pyenv/versions/3.7.4/lib/python3.7
/site-packages (from requests) (3.0.4)
Requirement already satisfied: beautifulsoup4 in /home/gimpei/.pyenv/versions/3.7.4/lib/python3.7/site-p
ackages (from bs4) (4.8.2)
Requirement already satisfied: soupsieve>=1.2 in /home/gimpei/.pyenv/versions/3.7.4/lib/python3.7/site-p
ackages (from beautifulsoup4->bs4) (1.9.5)
Installing collected packages: requests, bs4
  Running setup.py install for bs4 ... done
Successfully installed bs4-0.0.1 requests-2.22.0
```

図 x. インストール成功時に期待する画面

Pythonのファイルを書いて実行する

例えば、ヤフージャパンのサイトのタイトルをスクレイピングを試みると、以下のようなコードで実行することができます。

```
#!/usr/bin/env python3

import requests
from bs4 import BeautifulSoup

def main():
    r = requests.get('https://yahoo.co.jp')
    html = r.text
    soup = BeautifulSoup(html)
    print(soup.title.text)

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

このようなコードを実行すると、ヤフージャパンのトップページのタイトルの「Yahoo! Japan」というテキストが得られます。

では、ヤフー砲と呼ばれるぐらい影響力がある、ヤフーのトップのニュースのタイトルとリンクをスクレイピングするように上記のコードを改良してみましょう。

```

#!/usr/bin/env python3

import requests
from bs4 import BeautifulSoup
import re

def main():
    r = requests.get('https://yahoo.co.jp')
    html = r.text
    soup = BeautifulSoup(html)
    for a in soup.find_all('a', {'href':re.compile('https://news.yahoo.co.jp*')}):
        news_title = a.text
        link = a['href']
        print(news_title, link)
if __name__ == '__main__':
    main()

```

このコードで以下のような出力が得られました。

```

gimpei@Akagi:books_codes % python3 002.py
ゴーン被告拘束 ICPOが要請 https://news.yahoo.co.jp/pickup/6347105
ゴーン被告の旅券 地裁が許可 https://news.yahoo.co.jp/pickup/6347099
被災から25年 復興住宅に49% https://news.yahoo.co.jp/pickup/6347104
中国 北に「緊張高めるな」 https://news.yahoo.co.jp/pickup/6347102
新生児放置し死なす 母を逮捕 https://news.yahoo.co.jp/pickup/6347101
日本一太い 大アカマツ枯れる https://news.yahoo.co.jp/pickup/6347095
MAXのNANA 第1子出産を発表 https://news.yahoo.co.jp/pickup/6347103
紅白の歌唱 トリよりRAD長く https://news.yahoo.co.jp/pickup/6347097
もっと見る https://news.yahoo.co.jp/topics/top-picks?date=20200103&mc=f&mp=f
記事一覧 https://news.yahoo.co.jp/fc
ニュース https://news.yahoo.co.jp/

```

図 x. インストール成功時に期待する画面

これでヤフー砲を監視するスクリプトが書けますね。株価に重大な影響を及ぼすファクターだけに、いち早く察知することができるので他の人手の投資家に対して自動化などで先んじることができます。

BeautifulSoupはタグの種類とタグに与えられているプロパティのような要素で検索するようにアクセスすることができます。`soup.find_all('a', {'href':re.compile('https://news.yahoo.co.jp*')})` は、`<a>` のタグに対して `hrefで正規表現でhttps://news.yahoo.co.jp` に一致する範囲のタグをすべてリストで取り出す、という操作になります。

このタグはchromeのインスペクタで見たときと差があることに気づくと思います。実は、requestsではhttp, httpsで情報をくれというリクエストを投げるだけですのでJavaScript等の解釈ができません。つまり、JavaScriptが動くことで初めて描画されるようなコンテンツに関しては、全くのスルーになり、Google Chromeなどで見たときのhtml構造とは異なる事があるので、注意してください。Google ChromeなどでJavaScriptを停止するChrome拡張などを入れてhtmlの構造を最初に把握しておくと良いです。

2.2 Google Chrome + Seleniumでスクレイピングする

そもそもrequestsだけでスクレイピングが完結してしまうような構造のウェブサイトは多くのスクレイパーの餌食になると想像が付きます。

このとき、スクレイピングする側と、ウェブサイト側のスクレイピングされる側に利益相反などがあると、スクレイピング難易度を上げて防衛的な構造を取ることが多くあります。

requestsで取得する際にはJavaScriptが動作しないので、コンテンツの多くをJavaScriptに描画させるなどの手法が取られます。

2.2.1 Pixivを例に取る

数年前のPixivは割と簡単な構造で構築されており、簡単に殆どのイラストを収集することができましたが、現在はJavaScriptで多くのコンテンツをラップすることで、簡単には解析されないようにしています。

例えば、このようなコンテンツをユーザ側のブラウザで見ることができました。

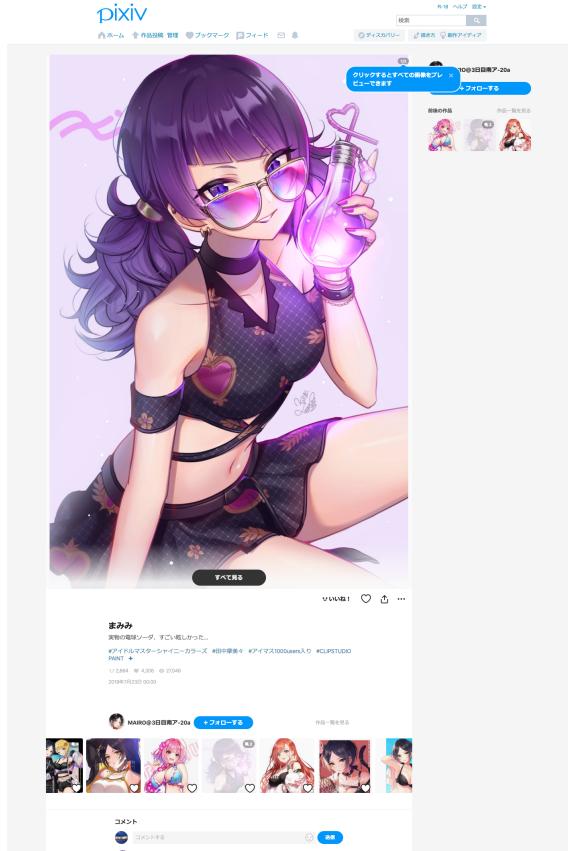


図 x. <https://www.pixiv.net/artworks/75863105>

では、このhtmlを解析しようとして、requestsでhtmlを取得して解析してみましょう。

```
# pixiv example only requests
import requests
from bs4 import BeautifulSoup

r = requests.get('https://www.pixiv.net/artworks/75863105')
soup = BeautifulSoup(r.text, 'html5lib')
for div in soup.find_all('div'):
    print(div)
```

期待としては、大量のdivタグの構造を取得できるはずですが、実際の2020年1月時点での出力は以下のようになります。

```
$ python3 006.py
<div id="root"></div>
```

このプログラムは卷末のgithubからダウンロードできます。

では、どうやってマミミのイラストを集めればいいのでしょうか？

シンプルで簡単な解決法としてGoogle Chromeをseleniumで動作させることで、期待する動作を得ることができます。

2.2.2 Pixivを例に取る: Google Chrome + Seleniumでデータを取得する

SeleniumとChromeDriverはインストール済みという前提で進めると、以下のコードでこのマミミのサイトのhtmlを取得できるはずです。

```
# headless google-chrome の例
import os
import shutil
from bs4 import BeautifulSoup
import time
import requests
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait

HOME = os.environ['HOME']
target_url = 'https://www.pixiv.net/artworks/75863105'
options = Options()
options.add_argument("--headless")
options.add_argument('window-size=2024x2024')
options.add_argument(f'user-data-dir={work_dir}')
options.binary_location = '/Applications/Google Chrome.app/Contents/MacOS/Google Chrome'

driver = webdriver.Chrome(executable_path=shutil.which('chromedriver'), options=options)
driver.get(target_url)
time.sleep(5.0)
html = driver.page_source
soup = BeautifulSoup(html, 'html5lib')
driver.save_screenshot("screenshot.png")
```

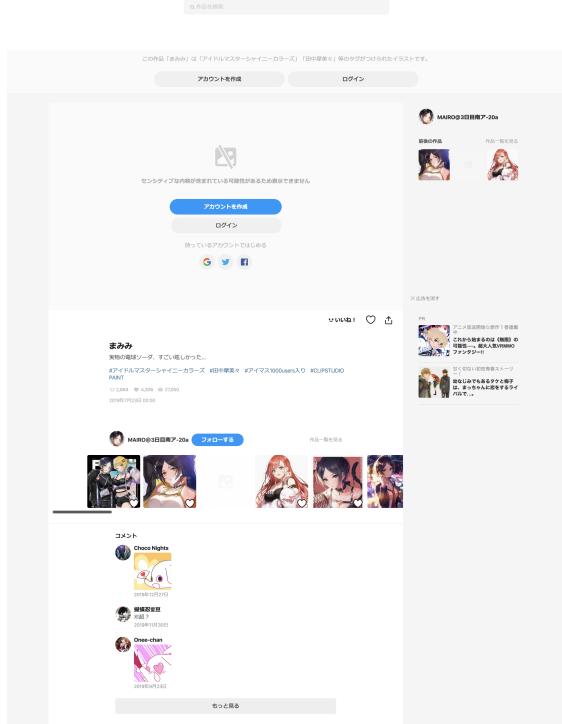


図 x. 結果

残念ながら、ログインしていないため、マミミは見ることができません。(どうして?)

次のステップでこれを更に拡張して、自動でログインして見るまで進めます。

2.2.3 Pixivを例に取る: 自動でログインしてデータを取得する

seleniumはユニークで他では代替ができない機能があって、その一つがブラウザを操作する機能です。

この機能を利用すると、自動でログインを行い、ログインのセッションが残っている間は、ログインしないと見えないはずのコンテンツを参照することができます。

以下のコードの例では、最初にログインを指定しない状態でGoogle Chromeを起動した後、ログインページにジャンプしてログイン項目を埋めて、起動します。

```
import os
import shutil
from bs4 import BeautifulSoup
import time
import requests
from pathlib import Path
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait

HOME = os.environ['HOME']
EMAIL = os.environ['EMAIL']
PASSWORD = os.environ['PASSWORD']

target_url = 'https://www.pixiv.net/artworks/75863105'
options = Options()
options.add_argument("--headless")
options.add_argument('window-size=2024x2024')
options.add_argument(f'user-data-dir={work_dir}')
options.binary_location = '/Applications/Google Chrome.app/Contents/MacOS/Google Chrome'

driver = webdriver.Chrome(executable_path=shutil.which('chromedriver'), options=options)
if not Path('init_chrome').exists():
    driver.get('https://accounts.pixiv.net/login')
    time.sleep(2.0)

elm = driver.find_element_by_xpath("//input[@autocomplete='username']")
elm.click()
elm.send_keys(EMAIL)
elm = driver.find_element_by_xpath("//input[@autocomplete='current-password']")
elm.click()
elm.send_keys(PASSWORD)
time.sleep(1.0)
elm = driver.find_element_by_xpath("//div[@id='LoginComponent']/button[@class='signup-form__submit']")
elm.click()
time.sleep(5.0)
Path('init_chrome').touch()

driver.get(target_url) # ここでまみみのページがログイン状態を保存してアクセスしてほしい
time.sleep(5.0)
html = driver.page_source
soup = BeautifulSoup(html, 'html5lib')
driver.save_screenshot("screenshot.png") # screenshotを取得する
```

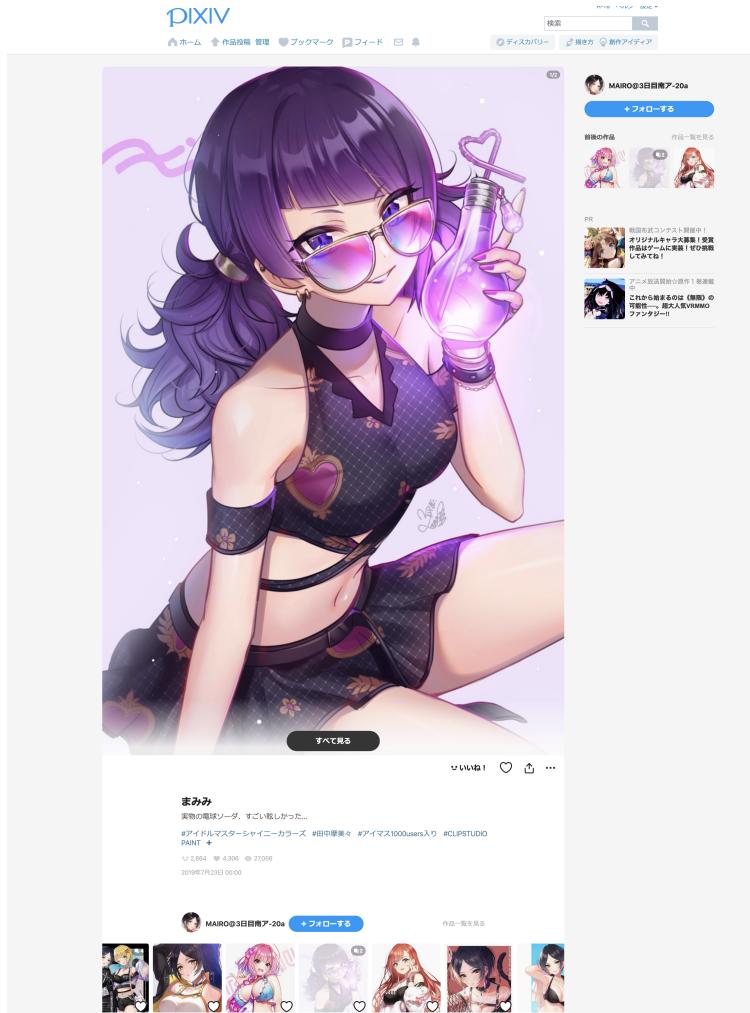


図 x. 最終的な出力のscreenshot.png, ただしマミミが表示されている

2.3 ハイパーリンクが作るネットワークは探索問題

ハイパーリンクはネットワーク状に繋がったネットワークをいかに探索するか、という問題にも帰着できます。

例えば以下のような図のネットワークがあったとします。

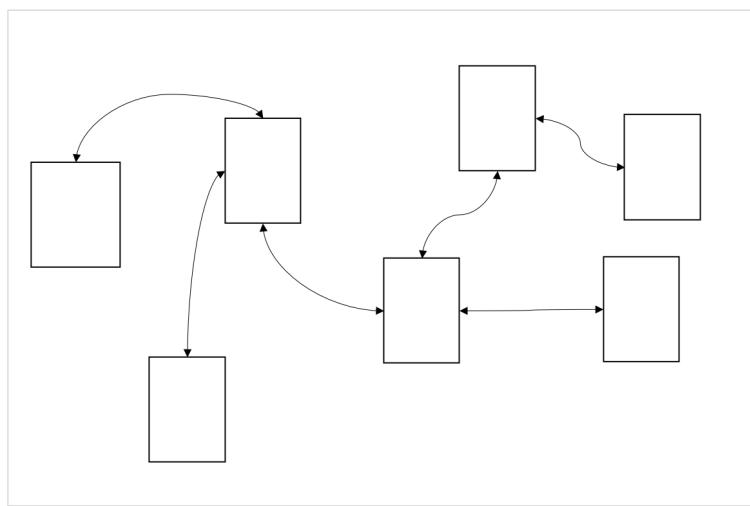


図 x. インターネットのhyper linkの依存関係の例

このとき、どこからどのようにスクレイピングすれば、サイト全体を取得することが可能になるのでしょうか。

これには考え方がいくつかあって、全取得を前提とした深さをあまり考慮しない方法をご紹介します。

まず、エントリーポイントとなる、pageを一つ決め、そのページをスクレイピングします。そのスクレイピングしたページ内部にあるURLを取り出して、次のページをスクレイピングします。何度も同じページをスクレイピングしてもサイトの負荷になるし、意味がないデータが増えるだけで

す。

Pythonをイメージしたコードで表現すると、以下のようになります。

```
#!/usr/bin/env python3
urls = [entry_url]
all_urls = set()
while True:
    for url in urls:
        if exists_db(url):
            continue
        html = get(url)
        store_db(url, html)
        for next_url in get_urls_from_html(html):
            if next_url not in all_urls:
                all_urls.add(next_url)
    urls = list(all_urls)
```

このようなコードが全探索を前提としたスクレイピングする際の最小のコードになります。

実際にはこのような理想通りに動作することはレアであり正しく動作させるために様々なヒューリスティックを入れることになります。

まずこのコードはシングルプロセスでしか動作させることができませんし、store_db, exists_dbで dbに大量にアクセスが発生します。

通常このようなユースケースではRDBは不向きで、例えばAWSのauroraのようなクエリ単位とスキャン量に対して課金されるようなDBの場合、一瞬で破産することが期待できます。

この問題を解決するには次の章の「3. 最強のDB、自作DBを作る」に記述します。

2.3.1 幅優先探索・深さ優先探索・ビームサーチ

ネットワーク状になっているので、探索方式がいくつか考えられます。

例えば幅優先探索でスクレイピングする場合、浅い領域を優先してスクレイピングを継続します。できるだけ外部ドメインに出ないようにコードを取得するなど向いています。

深さ優先探索になると、深い方を優先して探索するようになるのでネットワークから遠いところを優先してスクレイピングするようになります。URLが遠い場合などが有効です。

ビームサーチは幅優先探索と深さ優先探索のバランスを取ったような方法で、一定の探索幅を維持して、深さも幅も良いところどりをするものになります。

以下のコードの例では、幅優先探索で「yahoo.co.jp」のドメインをスクレイピングするものになります。

```
import requests
from bs4 import BeautifulSoup
import time
import re
from collections import namedtuple

DepthUrl = namedtuple('DepthUrl', ['depth', 'url'])
urls = [DepthUrl(0, 'https://www.yahoo.co.jp/')]
all_urls = set()
flatten_urls = set('https://www.yahoo.co.jp/')
depth = 0
for I in range(3):
    depth += 1
    for depth_-, url in urls:
        html = requests.get(url).text
        soup = BeautifulSoup(html, features="html.parser")
        for a in soup.find_all('a', {'href':re.compile(r'.*?\.\.yahoo\.co\.jp')})�:
            next_url = a.get('href')
            if next_url not in flatten_urls:
                all_urls.add(DepthUrl(depth, next_url))
        flatten_urls.add(url)
    all_urls -= {DepthUrl(depth_-, url)}
    urls = sorted(all_urls, key=lambda x:x[0])
min_depth = min([url.depth for url in urls]) #ここに注意
urls = [url for url in urls if url.depth == min_depth]
```

最も深さが浅いものをスクレイピングするように `min_depth` を算出していますが、ここを `max_depth` に変更したり、一定のルールでビーム幅を設定して計算量を抑えることでビームサーチにすることができます。

2.4 スクレイピングと法的問題

スクレイピングに関連する問題として常に隣接しているのは法的な問題点です。

何年も前の話ですが、図書館の在庫乗降を確認するサービスを作成しようとして、秒間1アクセスを満たすような低頻度のアクセスで通常は業務妨害に当たらないものであったのに関わらず、警察の厄介になってしまったという事件が起きました[X]

常識的なアクセスを行い、データ取得祭のサーバに高い負荷を送る意図がなくとも、このスクレイピングを認識する人物や組織やシステムの都合で逮捕、勾留されてしまうリスクを示すものでした。

基本的にはデータは提供しているサイトの個人や法人の持ち物であり、通常意図する用途での利用が想定されています。

例えば、何らかのデータを収集して機械学習等で法則性を取り出し、競合製品を作るなどは、業務妨害で訴えられても仕方がないと思います。一方で、そのサービスをより便利に、より利益がある形でAPIやGoogle Chrome拡張などを作るのは許容される範囲内だとだれも結果的に損していないので良い気がしています。

秒間1アクセスは近年のサーバリソースの拡充に伴い、現実的な基準という気はしていませんが、「対象サイトの業務をぼう会せず」に、可能ならば「相互に利益のある形」で分析やプロダクトづくりを行うといいでしょう。

- 岡崎市立中央図書館事件 :

<https://ja.wikipedia.org/wiki/%E5%B2%A1%E5%B4%8E%E5%B8%82%E7%AB%8B%E4%B8%AD%E5%A4%AE%E5%9B%B3%E6%9B%B8%E9%A4%A8%E>

3. 最強のDB、自作DBを作る

この章では、スクレイピングデータを保持したり可用性やSLAが高いDBをどう作っていくかを述べます。エンピリカルには、MySQLやPostgreSQLを用いるより、KVSなどより速度と非構造化データ(リレーションナル性が低いデータをこう呼びます)に対して、有効です。HTMLなどはベースをしてきれいに整形すれば、RDBなどと相性がいいですが、とりあえずベースや整形は後でデータを集めることを最優先するとKVS等が相性が良くなっています。

3.1 自作KVSとその関連

優れたKVSにはいくつか種類があり、有名なよくマネージメントされたKVSとしては、LevelDB, RocksDBなどがあります。

これらのDBは様々な例外を考慮されているし、速度も早いのですが、致命的なデメリットが存在して、マルチプロセスでのプロセス間を横断した、アクセスがきないという点があります。

逆に疎結合を意図したRBDのインターフェースなどはこの制約を受けないのですが、速度が極めてKVSと比較したときに遅くなります。

実践的で、実際に役立つKVSの作り方をご紹介したいと思います。

3.2 最強のDBは結局ファイルシステム

ファイルシステムという言葉は聞いたことがあるでしょうか。

ファイルシステムはOSやファイルを格納する際のSSDやHDDの、一定のルールで書き込み読み込みを制御するデータの格納法を示します。

馴染みがあるところだと、WindowsのNTFSや、MacOSのApple File Systemや、Linuxのext4などがあります。

いずれのファイルシステムも一長一短があり、たくさんのファイルを保存できるがデータ総量が少ない(xfs)、高速で動作して一定以上のパフォーマンスを発揮するが安定性が微妙(btrfs)、など多様性に富んでいます。

ファイルシステムは特定のキーに対して、計算量O(1)でアクセスできるので、実質KVSですし、この発想から作られたLevelDBはファイルシステムベースのKVSと言われています。

Pythonではファイルやバイナリデータのハッシュ値を扱うのが簡単であるため、簡単にKVSのようにファイルシステムを扱うことができます。

また、PythonではnamedtupleというkotlinなどのDataClassに相当するものがあり、シリализもサポートしています。

例として、なにかのkey, valueがある場合、O(1)でファイルシステムから書き込み、参照するコードを書いてみます。

通常のスクレイピングでは、何度も同じURLが出現する事があり、そのたびに新規リクエストを投げていたら、対象のサイトに異常な負荷を掛ける結果になってしまいます。

そのためダミーのURLを生成して、スクレイピングしたデータがあると仮定して、ファイルシステムを利用して、データを保存するコードを書いてみます。

```

# ファイルシステムベースのKVSの例
import gzip
import pickle
from pathlib import Path
from collections import namedtuple
from itertools import count
import requests
import random
from hashlib import sha224

Datum = namedtuple('Datum', ['depth', 'domain', 'html', 'links'])

def flash(url, datum):
    # keyとなるurlのhash値を計算(長過ぎるのトリムする)
    key = sha224(bytes(url, 'utf8')).hexdigest()[:24]
    # valueとなるDatum型のシリализと圧縮
    value = gzip.compress(pickle.dumps(datum))

    # db/にhash値のファイル名で書き込む
    with open(f'db/{key}', 'wb') as fp:
        fp.write(value)

def exists(url):
    # keyとなるurlのhash値を計算(長過ぎるのトリムする)
    key = sha224(bytes(url, 'utf8')).hexdigest()[:24]
    # もし、キーとなるファイルが存在していたら、それは過去にスクレイピングしたURLである
    if Path(f'db/{key}').exists():
        return True
    else:
        return False

dummy_urls = [f'{k}:04d' for k in range(1000)]
# ランダムなURLをスクレイピングしたとする
for i in range(10000):
    dummy_url = random.choice(dummy_urls)
    # すでにスクレイピングしていたURLならスキップする
    if exists(dummy_url) is True:
        continue
    depth = 1
    dummy_html = '<html> dummy </html>'
    dummy_domain = 'example.com'
    dummy_links = ['1', '2', '3']
    datum = Datum(depth=depth, domain=dummy_domain, html=dummy_html, links=dummy_links)
    flash(dummy_url, datum)

```

このコードはキーとなるURLに対して行った処理に対して、hash値をキーに、valueにnamedtupleを利用してKVS Likeなことをしています。

1000種類しかないURLに対して10000回も処理命令があった場合、重複するようなURLがあるはずでこのURLを効率的にスキップしたいです。その際に、hash名を持つファイルが存在するかどうかだけで判断するので、実質的にファイルシステムのアルゴリズムからこれはO(1)であることがわかります。

さてシンプルなこのコードを、並列化してみましょう。

この方法は、それぞれの並列化した関数（またはプロセス）から、ファイルシステムへアクセスすることが可能であり、例えばこれはLevelDBではバグってしまい完走できません。

```

# ファイルシステムベースのKVSの例
import gzip
import pickle
from pathlib import Path
from collections import namedtuple
from itertools import count
import requests
import random
from hashlib import sha224
from concurrent.futures import ProcessPoolExecutor

Datum = namedtuple('Datum', ['depth', 'domain', 'html', 'links'])

def flash(url, datum):
    # keyとなるurlのhash値を計算(長過ぎるのトリムする)
    key = sha224(bytes(url, 'utf8')).hexdigest()[:24]
    # valueとなるDatum型のシリализと圧縮
    value = gzip.compress(pickle.dumps(datum))

    # db/にhash値のファイル名で書き込む
    with open(f'db/{key}', 'wb') as fp:
        fp.write(value)

def exists(url):
    # keyとなるurlのhash値を計算(長過ぎるのトリムする)
    key = sha224(bytes(url, 'utf8')).hexdigest()[:24]
    # もし、キーとなるファイルが存在していたら、それは過去にスクレイピングしたURLである
    if Path(f'db/{key)').exists():
        return True
    else:
        return False

def parallel(arg):
    url, depth = arg
    if exists(url) is True:
        return
    depth = 1
    dummy_html = '<html> dummy </html>'
    dummy_domain = 'example.com'
    dummy_links = ['1', '2', '3']
    datum = Datum(depth=depth, domain=dummy_domain, html=dummy_html, links=dummy_links)
    flash(url, datum)

    dummy_urls = [f'{k:04d}' for k in range(1000)]
    # ランダムなURLをスクレイピングしたとする
    dummy_urls = [(random.choice(dummy_urls), i) for i in range(10000)]
    with ProcessPoolExecutor(max_workers=8) as exe:
        exe.map(parallel, dummy_urls)

```

ここで用いている `concurrent.future.ProcessPoolExecutor` は、引数に与えた関数をマルチコアで動作させるライブラリで、`max_worker` で並列数を指定できるので、この場合は8コアで動作することになります。
最近のCPUはコア数が多いのでリソースを最大に活かしながら、並列アクセス、並列バースなどができる、高速かつnon-blockingにKVSライクな処理を行うことができます。

なお、例えばLevelDBを使って並列処理を行おうと試みた場合、levelDBは使っているうちはロックが掛かるのと、ロックを無理やり上記のようなマルチプロセスライブラリ等で並列化した場合、DBのファイルに不整合が生じて、結果としてDBそのものを破壊してしまう、という事態になります。

```

$ python3 010.py | wc -l
Traceback (most recent call last):
File "010.py", line 55, in <module>
for key, value in db.iterator():
File "plyvel/_plyvel.pyx", line 362, in plyvel._plyvel.DB.iterator
File "plyvel/_plyvel.pyx", line 788, in plyvel._plyvel.Iterator.__init__
File "plyvel/_plyvel.pyx", line 94, in plyvel._plyvel.raise_for_status
plyvel._plyvel.Error: b'NotFound: /tmp/db//000005 ldb: No such file or directory'
1212

```

↑ LevelDBが並列処理により破壊された例。

4 . Depth 1, UserAgent, Referrer を偽装する

ブラウザには固有で、このブラウザの種類からアクセスしたよ、とこちらから宣言するためのheader情報が付きます。

`requests`だとこのように書くことでUserAgentを変更することができます

```

import requests
headers = {
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36'
}
r = requests.get('https://www.yahoo.co.jp/', headers=headers)

```

このようにすることで、Linux上で動作するPythonでアクセスしているはずですが、MacOS XのGoogle Chromeでアクセスしたことになります。

4.1 自分が使っているUserAgentを確認する

Googleで検索する際に、“my useragent”と入力すると、Googleが今使っているブラウザのUserAgentを教えてくれます。

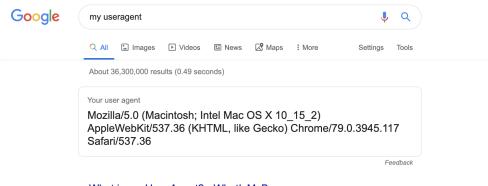


図. X Googleで今使っているブラウザのUserAgentを知ることができます

4.2 サイト管理人格に配慮する

サイト管理人格（個人や法人を想定しています）に対して、自分が行っているスクレイピング等の行為に対してフィードバックを得るチャンネルをUserAgentに組み込むことができます。

UserAgentやIPなどは、基本的にはアクセスログとして残り、サービスの改善に役立つので、分析対象になっていることが多く、ログに自らの意思を組み込むことで、平等性と透明性を確保することができます。

例えばこの例では、これが誰かのスクレーバーであること、サポートを得るためにURLを乗せるなどで、意図しないアクセスであっても、穩便に済ますことができます。

```

import requests
headers = {
    'User-Agent': 'Mozilla/5.0 (Linux; Analytics) [This is a scraper of nardtree"s analytics. https://gink03.github.io/]'
}
r = requests.get('https://www.yahoo.co.jp/', headers=headers)

```

4.3 Referrerを偽装する

今はこの要素がクローラーであるかどうかを判断する要素にはなりませんが、昔からあるレガシーなクローラーブロック術に、refereという前に何のページをみていたかで判断するサイトもあります。

この場合、requestsなどのアクセスにheader情報をつけることで、アクセスすることが可能になります。

refererに何かURLを入れるとそのURLから来たという意思表示になるというわけです。

```

# referrerを付ける例
import requests
from bs4 import BeautifulSoup

headers = {'referer': 'https://google.com'}
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36'
r = requests.get('https://www.yahoo.co.jp/', headers=headers)
soup = BeautifulSoup(r.text, features='html5lib')

```

5. Depth 2, IPを偽装する

最近のGoogle社のサービスや、GAFAなどの強いプレイヤーのサービスは機械学習によるBANが活発に行われていますが、今も現在もおそらく最強の不正利用（と彼らが定義する）の判別は、IPアドレスになります。

IPアドレスを偽装したり変えたりするのは、かなり重要で、一発アウト制をとっているサービスさんだと、復旧は事実上不可能になり、そのサービスに二度とアクセスできなくなります。

私の経験を踏まえつつ、どの程度まではやっていいのか、どこからはダメなのか、駄目ならどう駄目なのか、お伝えしていこうと思います。

5.1 プロキシサーバを立ててアクセス

最も簡単なのが何らかのクラウド業者にLinuxのインスタンスを借りて、プロキシサーバを立てて、そこ経由でアクセスすることです。

有名なプロキシソフトウェアは [squid](#) などがありこれをdockerで利用した際の利用法について述べます。

dockerはLinuxのコンテナという技術を用いて必要なソフトウェアの依存や環境などをまとめて、パッケージ化して、どのLinux環境でも再現できるようにしたものです。

例えば以下のコマンドで `squid` というソフトウェアのパスワード等の認証なしのプロキシサーバを建てることができます。

```
$ docker run -d -p 3128:3128 nardtree/squid
```

dockerが正しくインストールされていれば、これだけでOKです。とても簡単ですね。

これを利用して、curlコマンド等でアクセスするには以下のようになります。

```
$ curl --proxy http://user:user@133.130.97.98:3128/ https://ifconfig.co/  
133.130.97.98  
$ curl https://ifconfig.co/  
27.131.***.*** # オリジナルの私のIPが出ててしまう
```

IPアドレスは適宜インストールしたサーバのグローバルIPアドレスと読み替えてください。

単純な発想では、これをGCPやAWSやGMOクラウドやconohaの一番安価なインスタンスにデプロイしてIPを払い出せば、無限にアクセス制限が回避できることになります。しかし、本当にクラウド業者がそんなことを考えていないなどありうるのでしょうか？

pythonのrequestsでプロキシ経由で悪世するにはこのようなコードで実行する事ができます。

```
# proxy の例  
  
import requests  
  
proxies = {  
    "https": "http://user:user@133.130.97.98:3128/",  
    "http": "http://user:user@133.130.97.98:3128/"  
}  
r = requests.get('https://ifconfig.co/', proxies=proxies, verify=False)  
  
print(r.text)
```

5.1.2 閑話休題、AWSアカウントをバンされる

インセキュアなsquidなどを利用して、AWSで安いインスタンスにdockerを立ててプロキシサーバにしていると、当然、AWSとしては貴重なIPv4をブラックリストに入れられるリスクが増加するので許容できるものではありません。

ある日、私は個人で契約していたAWSがなんどインスタンスを立てても数秒でshutdownされてしまう謎の現象に遭遇しました。ドキュメントを漁っても該当する設定は存在しなく、有償サポートに賭けるしかない状況です。

いくらかの節制の結果、サポート一回分の費用を捻出した私は、有償サポートとは思えない冷たい一言を投げられます。

「不正利用の疑いがあるので、インスタンスの操作はできない。復旧の時期も、いつ利用再開になるのかも答えられない」

目の前が真っ暗になるようでした。

インセキュアで海外の攻撃者や第三者が利用した可能性が高いことが事態を余計に混迷させました。

幸いにして、趣味のwebサービスや分析用インスタンスは、GCP, AWSどちらに依存することなくdocker-composeで済む用なモジュール単位でwebサービスを作っていたので事なきを得ましたが、AWSのサービスに結合したものを作っていたり、動かす金額が大きいと個人で責任を取れなくなってしまいます。

5.2 公開プロキシ経由でのアクセス

全て自分でサーバを建ててアクセス、を行わなくても、Proxyサーバ自体は多くの人にいろんな政治的・思想的理由があり、それを補助するためにあらゆる地域と国家で建てられていたりします。

例えば、spys.one/en では、ロシアのサーバに様々なproxyが公開されています。

SPY.ONE/en/		Free proxy list	Proxy list by country	Anonymous free proxy	HTTPS/SSL proxy	SOCKS proxy list	Info	
Proxy by AS/NORC	Proxy by cities	Proxy by ports	HTTP proxy list	Transparent proxy list			Date	1
Free proxy list. Open proxy servers. Full unsorted list.								
							Show	30 ↗ ANM ↗ All ↗ SSL ↗ All ↗ Port ↗ All ↗ Type ↗ All ↗ Sort ↗ Date ↗ 1
Proxy address:port	Proxy type	Anonymity*	Country (city)	Hostname/ORG	Latency** Speed*** Uptime	Check date		
203.83.182.86:8080	HTTPS	(Mirrored)	NOA	Bangladesh (Dhaka)	203.83.182.86 (Grameen Cybernet Ltd. Bangladesh. AS for local peering and transit. Dhaka)	7.264 ↗ 51% ↗ 17-jun-2020 17:31	(198) ↗ 17-jun-2020 17:31	
159.192.104.53:8080	HTTP	(Mirrored)	NOA	Thailand	159.192.104.53 (CAT TELECOM Public Company Ltd.CAT)	8.39 ↗ 39% ↗ 17-jun-2020 17:31	(87) ↗ 17-jun-2020 17:31	
84.47.174.193:8080	HTTPS	(Mirrored)	NOA	Russia (Moscow)	84.47.174.193 (LLO Nauka-Svyaz)	12.089 ↗ 38% ↗ 17-jun-2020 17:31	(100) ↗ 17-jun-2020 17:31	
114.5.89.170:8080	HTTPS	(Mirrored)	NOA	Indonesia (Malang)	114.5-99.70.resources.indosat.com (INDOSAT Internet Network Provider)	4.65 ↗ 45% ↗ 17-jun-2020 17:31	(51) ↗ 17-jun-2020 17:31	
195.78.112.235:30544	HTTPS	HIA	Ukraine		195.78.112.235 (ProSC Value)	1.695 ↗ 14% ↗ 17-jun-2020 17:30	(14) ↗ 17-jun-2020 17:30	
176.120.215.102:8080	HTTP	(Mirrored)	NOA	Russia (Makhachkala)	176.120.215.102 (Subnet LLC)	2.283 ↗ 47% ↗ 17-jun-2020 17:30	(35) ↗ 17-jun-2020 17:30	
200.215.171.238:8080	HTTPS	(Mirrored)	NOA	Brazil (Recife)	host-171-238.smart.net.br (SMART TELECOMUNICAÇÕES E SERVIÇOS EIRELI EPP)	11.256 ↗ 54% ↗ 17-jun-2020 17:29	(35) ↗ 17-jun-2020 17:29	
202.57.2.19:8080	HTTP	(Mirrored)	NOA	Indonesia (Bekasi)	202.57.2.19 (PT. Khasnah Timur Indonesia)	3.451 ↗ 39% ↗ 17-jun-2020 17:28	(89) ↗ 17-jun-2020 17:28	
151.80.65.175:3128	HTTP	(Mirrored)	NOA	France (Roubaix)	151.80.65.175 (OVH SAS)	8.533 ↗ 87% ↗ 17-jun-2020 17:28	(99) ↗ 17-jun-2020 17:28	
42.112.209.164:8080	HTTP	(Mirrored)	NOA	VietNam (Hanoi)	42.112.209.164 (The Corporation for Financing & Promoting Technology)	9.556 ↗ 22% ↗ 17-jun-2020 17:28	(99) ↗ 17-jun-2020 17:28	
151.80.4.233:3128	HTTP	(Squid)	NOA	Germany	15253-151-80-4.eu (OVH SAS)	0.199 ↗ 45% ↗ 17-jun-2020 17:28	(48) ↗ 17-jun-2020 17:28	
69.65.65.178:34546	HTTP	HIA	United States (Vermont Beach)		crisp-69.65.178.myocean.net (Blue Stream)	4.66 ↗ 82% ↗ 17-jun-2020 17:28	(100) ↗ 17-jun-2020 17:28	
149.56.191.12:8080	HTTP	(Squid)	NOA	Canada (Montreal)	ip12.ip-149-56-191.net (OVH SAS)	0.614 ↗ 53% ↗ 17-jun-2020 17:28	(146) ↗ 17-jun-2020 17:28	
149.28.240.236:3128	HTTP	ANM	United States (Dallas)		149.28.240.236.vfrt.com (Chopra, LLC)	0.777 ↗ 41% ↗ 17-jun-2020 17:28	(16) ↗ 17-jun-2020 17:28	
147.153.3.105:999	HTTPS	(Mirrored)	NOA	United States (Reston)	micro.cloudnet.uy (OVH SAS)	3.485 ↗ 80% ↗ 17-jun-2020 17:28	(100) ↗ 17-jun-2020 17:28	
222.127.15.81:3128	HTTP	NOA	Philippines		222.127.15.81 (Globe Telecoms)	1.679 ↗ 91% ↗ 17-jun-2020 17:27	(105) ↗ 17-jun-2020 17:27	
203.76.124.35:8080	HTTP	(Mirrored)	NOA	Bangladesh	203.76.124.35 (Link3 Technologies Ltd.)	2.767 ↗ 82% ↗ 17-jun-2020 17:27	(142) ↗ 17-jun-2020 17:27	
150.107.205.198:8081	HTTP	NOA	Nepal		150.107.205.198 (TECHMINNOS PVT. LTD.)	3.01 ↗ 57% ↗ 17-jun-2020 17:28	(100) ↗ 17-jun-2020 17:28	
67.75.2.39:3128	HTTP	NOA	United States (Miami)		67.75.2.39 (Level 3 Parent, LLC)	1.673 ↗ 80% ↗ 17-jun-2020 17:28	(669) ↗ 17-jun-2020 17:28	
200.60.13.122:999	HTTP	(Mirrored)	NOA	Peru	200.60.13.122 (Telefonica de Peru S.A.A.)	3.827 ↗ 10% ↗ 17-jun-2020 17:28	(10) ↗ 17-jun-2020 17:28	
140.227.176.104:80	HTTP	ANM	Japan		140.227.176.104 (NTT PC Communications, Inc.)	1.075 ↗ 100% ↗ 17-jun-2020 17:28	(100) ↗ 17-jun-2020 17:28	
14.207.77.131:8080	HTTP	(Mirrored)	NOA	Thailand (Prasae)	mx-8-14-207.77-131.dynamic.3bb.in.th (Triple T Internet/Triple T Broadband)	4.505 ↗ 100% ↗ 17-jun-2020 17:28	(100) ↗ 17-jun-2020 17:28	
110.44.117.26:43922	HTTP	HIA	Nepal		110.44.117.26 (VarNet Communications Pvt. Ltd.)	3.303 ↗ 92% ↗ 17-jun-2020 17:28	(139) ↗ 17-jun-2020 17:28	
14.207.24.160:8080	HTTP	(Mirrored)	NOA	Thailand (Rayong)	mx-8-14-207.24-160.dynamic.3bb.co.th (Triple T Internet/Triple T Broadband)	1.82 ↗ 89% ↗ 17-jun-2020 17:28	(100) ↗ 17-jun-2020 17:28	
193.66.135.123:59278	HTTPS	HIA	Bulgaria		dispy123-135.cip.digrs.bg (Digital Systems Ltd.)	1.65 ↗ 38% ↗ 17-jun-2020 17:24	(23) ↗ 17-jun-2020 17:24	
193.89.118.59:8080	HTTP	(Mirrored)	NOA	Thailand (Pattaya)	mx-8-183.89.118-59.dynamic.3bb.co.th (Triple T Internet/Triple T Broadband)	1.806 ↗ 100% ↗ 17-jun-2020 17:24	(31) ↗ 17-jun-2020 17:24	
63.221.178.32:8080	HTTP	(Mirrored)	NOA	Uzbekistan	32.178-221-83.stream.uz (Shang Telekom CJSC)	2.824 ↗ 90% ↗ 17-jun-2020 17:23	(60) ↗ 17-jun-2020 17:23	

図.X spyx.one/en

このサイトでは、常にフレッシュなプロキシサーバが公開されて、IPバンリスクが高く、IPがすぐ何らかの理由でバンされてしまったとしても、すぐ次のプロキシに乗り換えることができます。

このバンされたらIPを乗り換える、ということを自動化しようとすると、実はハードルが一箇所あります。

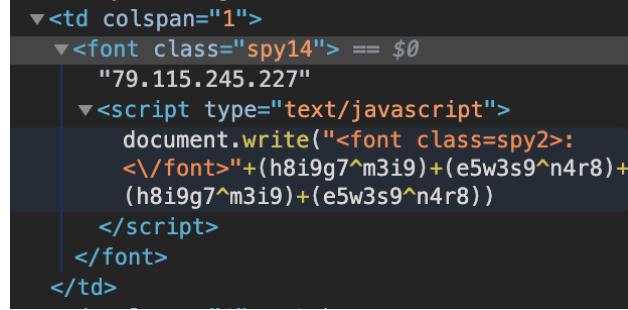


図.X Proxyのポートの表示が実は、JavaScriptの演算で得られる

JavaScriptでPortが描画されている、かつ、変数の定義がこの面だけは解析的に出すのが難しめの競プロのようになってしまっています。

概して、このような障害がある際に、有効な方法はchromeのheadlessブラウザによるアクセスであり、JavaScriptを実行させてからhtmlをベースとするという方法が有効です。

以下にchromedriverを用いて、google-chromeをheadlessで動作させて、[spys.one/en] からプロキシのリストを取得するコードを例示します。

```

from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.by import By
import time
import re
import requests
from bs4 import BeautifulSoup
import json

options = Options()
options.add_argument('--headless')
options.add_argument("user-agent=Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36")
options.add_argument(f"user-data-dir=/tmp/work")
options.add_argument('lang=ja')
options.add_argument('--disable-dev-shm-usage')
options.add_argument("--window-size=1080,1080")
driver = webdriver.Chrome(options=options,
                           executable_path='/usr/bin/chromedriver')

proxies = set()
for proxy_src in ['http://spys.one/free-proxy-list/JP/', 'http://spys.one/en/free-proxy-list/']:
    driver.get(proxy_src)
    time.sleep(1.0)
    driver.find_element_by_xpath(
        "/select[@name='xpp']/option[@value='5']").click()
    time.sleep(1.0)
    html = driver.page_source
    soup = BeautifulSoup(html)
    [s.extract() for s in soup('script')]
    #print(soup.title.text)
    for tr in soup.find_all('tr'):
        if len(tr.find_all('td')) == 10:
            tds = tr.find_all('td')
            ip_port = tds[0].text.strip()
            protocol = re.sub(
                r'\.(.*?\.)*', "", tds[1].text.strip().lower().strip())
            proxy = f'{protocol}://{ip_port}'
            proxies.add(proxy)
proxies = list(proxies)

with open('proxies.json', 'w') as fp:
    json.dump(proxies, fp, indent=2)

```

5.3 tor経由でのアクセス

よほどのことがない限りtorは使う機会が無いのですが、ダークウェブを定量的な視点で分析するなどのモチベーションがあればユースケースとして最適です。

tor経由でアクセスするとほぼ通信元がたどれなくなるという匿名性があります。

dockerを用いたサーバの立て方、環境変数を用いたアクセスの仕方、pythonのモジュール内で指定して悪世する方法をお伝えします。

5.3.1 Dockerでtorのsocks5 proxyサーバを立てる

socks5 というプロトコルでtorネットワーク経由でアクセスするproxyサーバを建てることができます。

サーバとなるLinuxなどで、以下のコマンドを入力するだけでサーバーが立ち上ります。

```
$ docker run -d --restart=always -p 0.0.0.0:9150:9150 peterdavehello/tor-socks-proxy:latest
```

conoha(IP:133.130.97.98)というクラウドサービスにサーバを建てて自宅のMac(IP:126.140.215.0)からアクセスして、IPアドレス確認サービスにIPを問い合わせると以下の結果が帰ってきました。

```
$ curl --socks5-hostname 133.130.97.98:9150 https://ipinfo.tw/ip
209.95.51.11
```

このIPを `whois` で確認すると以下のような結果が得られ、全く関係のないIP扱っていることがわかります。 (headerの特定可能な情報も削っています)

```

OrgName: Hosting Services, Inc.
OrgId: HOSTI-20
Address: 517 W 100 N STE 225
City: Providence
StateProv: UT
PostalCode: 84332
Country: US
RegDate: 2008-03-03
Updated: 2017-02-08
Ref: https://rdap.arin.net/registry/entity/HOSTI-20

```

5.3.2 環境変数でsocks5を指定する

環境変数に設定して用いるとproxyを広いレンジで用いることができて便利です。
torのプロトコルはsocks5hというものになっていて、curlでは用いることができるが、wgetはできないなどの微妙な成約があります。

```

$ export http_proxy=socks5h://133.130.97.98:9150
$ export https_proxy=socks5h://133.130.97.98:9150
$ export all_proxy=socks5h://133.130.97.98:9150

```

5.3.3 Pythonでsocks5を指定する

requestsで用いるとき socksのサポートをpythonで行うため、このようなモジュールをインストールする必要があります。

```
$ pip install "requests[socks]"
```

pythonは以下のようなコードが期待されます。

```

import requests
proxies=dict(http='socks5h://133.130.97.98:9150',
            https='socks5h://133.130.97.98:9150')
r = requests.get('https://ipinfo.tw/ip', proxies=proxies)
print(r.text)

```

出力は `109.70.100.30` でした。このIPの所有者は `TOR-EXIT-FOUNDATION-FOR-APPLIED-PRIVACY` ですので、無事隠蔽されました。

6. Depth 3, MultiCore, Multi Machineでスクレイピングする

現代のモダンなコンピュータはCPUを複数備えることが一般的になっています。

通常のPythonスクリプトはSingle CPUしかリソースを使えませんが、大量のCPUのコア（スレッドと表現することもありますがコアと統一します）を効率的に使える標準ライブラリと動作原理をお伝えします。

- 1. concurrent.futures.ProcessPoolExecutor
- 2. concurrent.futures.ThreadPoolExecutor
- 3. asyncio

よく整理されて使い勝手が良いライブラリがこちらになります。

ProcessPoolExecutorがマルチプロセスで、ThreadPoolExecutorがGILというレガシーな仕組みをつかったスレッド、asyncioがノンブロッキングIOという方法で動作するスレッドになります。

マルチプロセスとマルチスレッドは大学でコンピュータサイエンスをやった人には、积遡に説法ですが、そうでない人のために説明させていただけます。

マルチプロセスは、forkという機能を使って実行しているプログラムのOSから見る実態を増やします。このときメモリに重なる部分が多いので消費を低減して、実態を複数個作ります。forkで新たに作られた子プログラムと親プログラムは原則としてなんらか細工をしない限り通信できません。

マルチスレッドは、一つのプログラムの中で、CPUリソースを割り当てを変えながら同時に2つ異常動作せる仕組みです。このCPU割あてスケジューリングには色々手法があって、ioがブロックされている間は別の処理をするようにしたものがasyncioが行うスレッドになります。

これらの特性をまとめると以下のようになります。

表.X 特性比較

	MultiProcess	Thread
メモリ効率	悪い	良い
速度	良い	悪い
共有変数のシンク	原則できない	できる

つまりユースケースに応じて、並列化の手法を使い分ければ良いとわかります。

例えば今回の主ミッションであるスクレイピングにたいしては、ThreadとMultiprocessingはどちらが良いのでしょうか？

6.1 Thread vs Multiprocessing

では、実際にスクレイピングの文脈に対しては、ThreadとMultiprocessingはどちらが効果的に動作するのでしょうか。

実際にユニークなドメイン [2941件] のURLに全量をスクレイピングするのにどの程度かかるかを示します。

スクレイピングの速度をベンチマークするに当たって、ドメインを限定するは不適切ですので、ドメインでユニークにすることで負荷を分散しました。

なお、使っているwrapperが `concurrent.futures.ProcessPoolExecutor` か `concurrent.futures.ThreadPoolExecutor` の違い飲みになっているのでここでは、Multiprocessの方のコードだけを例示します。

必要に応じて、巻末のgithubのリンクを参照してください。

```
from urllib.parse import urlparse
from pathlib import Path

from bs4 import BeautifulSoup
from tqdm import tqdm
from urllib.parse import urlparse
from concurrent.futures import ProcessPoolExecutor
import pickle
import random
import requests
import time
with open('002.pkl', 'rb') as fp:
    netloc_urls = pickle.load(fp)

urls = []
for netloc, _urls in netloc_urls.items():
    if len(_urls) >= 10:
        url = random.sample(list(_urls), k=1)[0]
        urls.append(url)

print('total uniq domain(netloc) url size is', len(urls))

def parallel(arg):
    try:
        url = arg
        print(url)
        r = requests.get(url, timeout=5.0)
        soup = BeautifulSoup(r.text, 'html5lib')
        hrefs = set()
        for a in soup.find_all('a', {'href':True}):
            a.get('href')
        return hrefs
    except Exception as exc:
        print(exc)
        return set()

start = time.time()
hrefs = set()
with ProcessPoolExecutor(max_workers=10) as exe:
    for child_hrefs in exe.map(parallel, urls):
        hrefs |= child_hrefs
elapsed = time.time() - start
print(f'elapsed time {elapsed}')
```

MultiProcessingの結果

```
$ python3 003-multiprocess.py
elapsed time 496.5804433822632
```

単位は秒になります。

Threadingの結果

```
$ python3 003-threading.py
elapsed time 1580.4620769023895
```

単位は秒になります。

このように、基本的に戻り値だけで制御できたり重い操作をやる場合は、Multiprocessingの方が速度的に優れていることがわかりました。

実はMultiprocessingの真価は、全CPUを効率的に用いられるというだけではありません。ロジックの組み方によってはマルチコアのCPU内部に閉じずに、マシンを横断しての並列処理ができるこことを次の章で示します。

6.2 MultiprocessingをMulti Machineに拡張

Multiprocessingは原則として（例外の非常に多い原則ですが）プロセス間でメモリ内容の共有ができません。

一見不便なようこの制約ですが、Linux, UnixのPIPEやhttp通信やファイルシステムをバイパスすればこれらの制約は回避することができます。

Multiprocessingの1プロセスがアクセスする先をファイルシステムにすることで、PIPEや、httpなどを用いなくても簡単かつ大規模な共有メモリプールとして利用できます。

例えば、オーバーヘッドを避けるために、すでにスクレイピングしたURLのキーが共有フォルダー上に存在したら処理をスキップするなど簡単に組めます。

例えばデータをホストするマシンを一台組み、nfsやsshfsなどで、リモートのマシンのフォルダーやハードディスクを共有すると同時に並列にアクセスできるようになります。

スクレイピングしたurlをhtml等以外にもlinkをjson等で保存すれば、そのjsonファイルを別のマシンで読み書きできるようになります。

このコードはnfsでマウントしたSSD等から起動すると、どのマシンでも並列で実行できるよになります。

```

import requests
from bs4 import BeautifulSoup
from concurrent.futures import ProcessPoolExecutor as PPE
from urllib.parse import urlparse
from hashlib import sha224
from pathlib import Path
from multiprocessing import Process, Manager
import random
import json
import glob
from tqdm import tqdm
manager = Manager()
domain_freq = manager.dict()

def run(arg):
    i, url = arg
    mst_p = urlparse(url)
    mst_netloc = mst_p.netloc
    group_key = mst_netloc
    key = sha224(bytes(url, 'utf8')).hexdigest()[:24]
    if Path(f'htmls/{group_key}/{key}').exists():
        return set()
    if Path(f'errs/{group_key}/{key}').exists():
        return set()
    try:
        r = requests.get(url, timeout=30)

        if mst_netloc not in domain_freq:
            domain_freq[mst_netloc] = 0

        all_count = sum(domain_freq.values())
        if all_count > 10 and random.random() < domain_freq[mst_netloc]/all_count:
            return set([url])
        domain_freq[mst_netloc] += 1

        mst_scheme = mst_p.scheme
        soup = BeautifulSoup(r.text, 'html5lib')

        next_urls = set()
        for tag in soup.find_all('a', {'href': True}):
            href = tag.get('href')
            if 'javascript' in href:
                continue
            p = urlparse(href)
            if p.netloc == '':
                p = p._replace(scheme=mst_scheme, netloc=mst_netloc)

            p = p._replace(params='')
            p = p._replace(query='')

            href = p.geturl()
            next_urls.add(href)

        Path(f'htmls/{group_key}').mkdir(exist_ok=True, parents=True)
        with open(f'htmls/{group_key}/{key}', 'w') as fp:
            fp.write(r.text)
        Path(f'links/{group_key}').mkdir(exist_ok=True, parents=True)
        with open(f'links/{group_key}/{key}', 'w') as fp:
            fp.write(json.dumps(list(next_urls)))

        return next_urls
    except Exception as exc:
        print(exc)
        Path(f'errs/{group_key}/{key}').mkdir(exist_ok=True, parents=True)
        Path(f'errs/{group_key}/{key}').touch()
        return set()

args = [(0, 'https://news.yahoo.co.jp/pickup/6348371')]
while True:
    next_urls = set()
    with PPE(max_workers=32) as exe:
        for _next_urls in exe.map(run, args):
            next_urls |= _next_urls
    if len(next_urls) == 0:
        fns = glob.glob('links/*/*')
        random.shuffle(fns)
        for fn in tqdm(fns):
            next_urls |= set(json.load(open(fn)))

```

```

print('do')
args = [(i, url) for i, url in enumerate(next_urls)]

```

この方法での並列化は2～10台程度では、ほとんど線形に性能が向上するが多く、気軽にハイパフォーマンス・コンピューティングをするのに向いている方法になります。

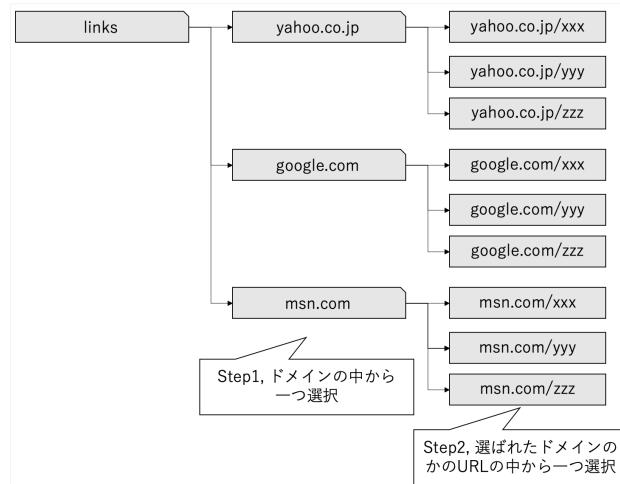
7. フェアネスを考慮したスクレイピング

AWSやGCPで特定のとメインに対して高頻度でアクセスを行うと攻撃とみなされ、通信がシャットダウンされたりします。また実際にアクセスが多すぎて、対象のサイトに対して過負荷を起こしてしまってもかもしれません。

いくつかのロジックにて過負荷を防ぐ方法があるので、ご紹介するとともに、どういったロジックで作成したかを説明します。

7.1 ランダムドメイン選択

結局の所、攻撃とみなされる要素は特定のドメインに対して大量のリクエストを行うことで判定されるので、負荷を分散する必要があります。もっと簡単なロジックでは、すべてのドメインに対して平等なスクレイピングの機会を与える方法で、以下のような構造のファイルシステムがある場合に、まず階層1のfolderからドメインを選択し、そのドメインのfolderないのURLを一つ選択し、そのURLを消去します。この操作をすべてのfolderが空になるまで繰り返し、空になったらスクレイピング終了です。（現実的な実世界のウェブページ数はものすごい数にのぼり、終了条件を満たせることはほぼありません）



図x. ランダムドメイン選択

この手法はすべてのURLに対して平等な負荷をかけることになるので、現実的に誰かに迷惑をかけるということはありません。しかしながら、特定のどのdomainに注力をするということもしないので、無限に薄く広く広がることになります。（これを防ぐには日本語のドメインに限定するなどすると良いです）

8. 練習

この章では、いくつかの目的を設定してそれぞれのユースケース別に同スクレイピングをしていくのか実演形式でお伝えします。

8.1 Practice 1, 無料のphotostockをスクレイピングして大量のフリー画像を集め

いくつかのサイトではphotostockと呼ばれる無料の写真を提供していることがあります。ブログのイメージ画像に設定したり、タグやなにかの属性が付与されれば、機械学習に用いることができたりなど、汎用性が高いサービスです。

何らかの理由により、これらのデータが必要になった場合、どのような戦略で集めることができるかを述べます。

8.1.1 https://unsplash.com/

対象とするサイトは unsplash.com としました。

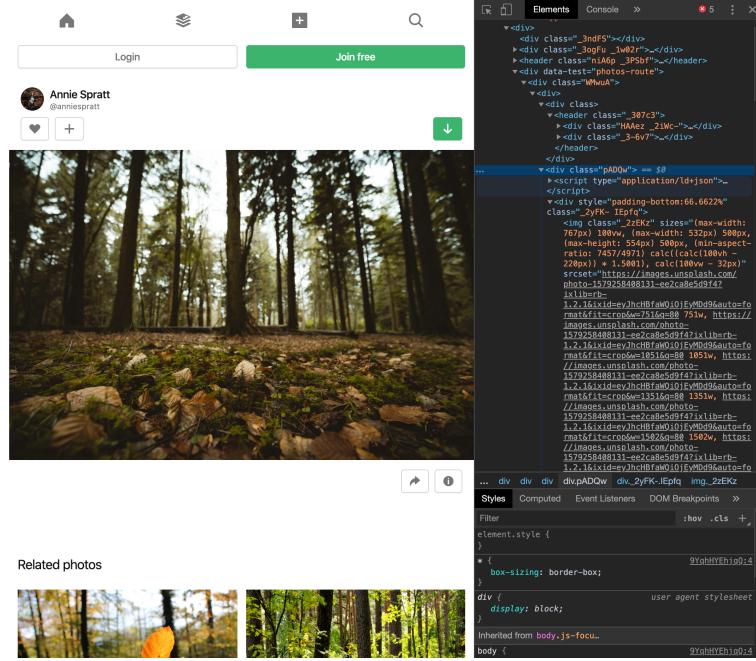
このサイトは画像を大量に公開しているサイトで、商用利用等を含めて競合しない限り自由に利用してOKというすごい太っ腹なサイトです。

構造的にはシンプルですが、htmlの作りが機械的で法則性が弱く一部ヒューリスティックとアドホックを入れて対応することになります。

例えば、`div` の classやidが唯一に特定できればそれが一番ありがたいのですが、特に指定がない構造であったり、では、順番によって特定できるかというと、ページによって再現性がなかったりなど、行き着くところはルールを発見し（ヒューリスティック）、場当たり的に対応（アドホック）するしかないです。

ウェブページに限らず、データがきれいに整備されているとき、ヒューリスティックとアドホックが多く入るとき、そのデータは完全性や完成度が低いとみなすことができますが、実際のデータ構造に近いので、練習になります。

小さいコードを組んで、目的に対して動作するかを検証することから始めます。このときchromeやvivaldiなどで、タグ構造を把握しながら、どのような方針が適切か検討します。



様々な角度でこのウェブサイトの構造を検証したところ、以下のことが判明しました。

- divのclass名がページごとのユニークなhashが与えられており、指定することができない
- 構造に多様性があるため “<https://unsplash.com/photos>” で始まるURLに限定して見たほうがいい
- 今回ほしい画像とその説明は最も大きい画像であり、その画像のみalt属性が存在し、それがフラグになりそうである
- “<https://unsplash.com/photos/.{1,}/download>” のURLを踏んでしまうと応答が帰ってこなく、requestsがtimeoutしてしまうので、このURLは避ける

など、多くのこのドメイン限りのルールが判明しました。

以下にコードを記します。

```
from concurrent.futures import ProcessPoolExecutor as PPE
import glob
import requests
from bs4 import BeautifulSoup
from urllib.parse import urlparse
from hashlib import sha224
from pathlib import Path
import random
import json
import gzip
from tqdm import tqdm
import re
from multiprocessing import Process, Manager
import time

manager = Manager()
shared_d = manager.dict({'requests':0, 'bs4':0, 'img_dl':0, 'href':0 })

def hashing(url):
    x = sha224(bytes(url, 'utf8')).hexdigest()[:16]
    return x

def parallel(arg):
    try:
        url = arg
        mst_p = urlparse(url)

        mst_p = mst_p._replace(query="")
        url = mst_p.geturl()
        if re.search('download$', url):
            return set()
        if 'https://unsplash.com/photos' not in url:
            return set()

        print(url)
        start = time.time()
        with requests.get(url) as r:
            html = r.text
            shared_d['requests'] += time.time() - start

        start = time.time()
        soup = BeautifulSoup(html, 'lxml')
        shared_d['bs4'] += time.time() - start

        mst_x = hashing(url)
        if Path(f'htmls/{mst_x}').exists():
            return set()

        start = time.time()
        for img in soup.find_all('img', {'src': re.compile('https://images.unsplash.com/photo'), 'alt':True}):
            src = img.get('src')
            alt = img.get('alt')
            p = urlparse(src)
            p = p._replace(query="")
            src = p.geturl()
            name = hashing(src)
            #print('query removed url', src, 'hashing', name)
            if Path(f'imgs/{mst_x}/{name}.jpg').exists():
                continue
            print('try download', src, alt, 'at', url)
            with requests.get(src) as r:
                binary = r.content
            Path(f'imgs/{mst_x}').mkdir(exist_ok=True, parents=True)
            with open(f'imgs/{mst_x}/{name}.jpg', 'wb') as fp:
                fp.write(binary)
            with open(f'imgs/{mst_x}/{name}.txt', 'w') as fp:
                fp.write(alt)
            shared_d['img_dl'] += time.time() - start

        start = time.time()
        hrefs = set()
        for a in soup.find_all('a', {'href': True}):
            href = a.get('href')
            try:
                if href[0] == '/' or 'https://unsplash.com' in href:
                    if href[0] == '/':
                        href = 'https://unsplash.com' + href
                    hrefs.add(href)
            except:
                continue
    
```

```

except Exception as exc:
    print(exc)
shared_d['href'] += time.time() - start

x = hashing(url)
with Path(f'links/{x}).open('w') as fp:
    json.dump(list(hrefs), fp, indent=2)
with Path(f'htmls/{x}).open('wb') as fp:
    ser = gzip.compress(bytes(html, 'utf8'))
    fp.write(ser)

print(shared_d)
return hrefs

except Exception as exc:
    print('exc', exc, url)
    return set()

urls = ['https://unsplash.com/photos/D1IS5s5O9xo']
while True:
    nexts = set()
    with PPE(max_workers=24) as exe:
        # for url in urls:
        #     nexts |= parallel(url)
        for hrefs in exe.map(parallel, urls):
            nexts |= hrefs

    if len(nexts) == 0:
        nexts = set()
        for fn in tqdm(glob.glob('links/*')):
            try:
                with open(fn) as fp:
                    nexts |= set(json.load(fp))
            except:
                continue
        urls = list(nexts)
    else:
        urls = list(nexts)

```

8.3 Practice 3, YouTubeの動画をダウンロードする

誰もがチャレンジしたくなるYouTubeのダウンロードですが、YouTubeはGoogle社に買収されてからというもの、機械学習による高速な異常検知による自動BAN等で、ダウンロードができる手法だったり穴が発見されても、なかなか利用できないというのが現状になります。

異常検知に引っかかると、IPごとBANになるのでこれを避けながらダウンロードする術があるので、ご紹介したいと思います。

8.3.1 YouTubeの動画のダウンロードに便利な youtube-dl

chromedriverでhtmlを解析して動画のstream通信をディスクに保存する手法を最初は検討していたのですが、YouTubeというサイトが高速に仕様やセキュリティの様相を変化させて、キャッチアップしきれず、ライブラリに頼ることになりました。

もっともダウンロード成功率が高いのがpipで入る youtube-dl というソフトで、最新のYouTubeの仕様に対応するため、github経由でインストールすると確実です。

```
$ pip install git+https://github.com/yt-dlp/yt-dlp
```

具体的に、動画をダウンロードするには以下のコマンドが必要です。

```
$ youtube-dl "https://www.youtube.com/watch?v=xxxxx" # xxxxはサンプル
```

いろんな動画をダウンロードできるのですが、何回か使っていると気づくはずです。。。突然、ダウンロードできなくなることに。

この状態になってしまふとしばらく待ってもなかなか解除されませんが、MacBookなどのChromeでYouTubeを見るぶんには大丈夫だったりします。

8.3.2 IPを使い潰すしかなさそう

“IPレベル”×“ツールレベル”で規制されることがわかっているので、IPアドレスを無限に変え続けなければいいという単純な発想に落ち着きます。

以前の章で紹介した <http://spys.one/en> のプロキシサーバリストを使うことができます。

どれか回線状態がよいProxyで以下のコマンドを実行すると、Proxy経由のアクセスになり、ダウンロードを継続することができます。

```
$ youtube-dl --proxy 118.27.31.50:3128 "https://www.youtube.com/watch?v=xxxxxx"
```

`spys.one/en` を更に別の章で紹介したgoogle chrome + seleniumでのダウンロードを用いることにより、Proxyの取得すら自動化できるので、ほぼ制限なしに使えるようになります。

※ すべての動画と音声がダウンロード可能ではありますが、法で規制されているコンテンツも多いので、十分に留意してください。

10. 閑話休題2, リクエストが多すぎるとDNSが応答しなくなる

膨大な通信を行うと、Googleの [8.8.8.8](#) やCloudFlareの [1.1.1.1](#) などにアクセスしてドメインを解決するのはかなりのコストであり、家や会社のルータでは膨大なリクエストを捌くにあたってすべてをGoogleやCloudFlareに投げると、だんだんDNSが応答してくれなくなってしまいます。

そのため、自前で簡単なDNSサーバを立てたほうがいいのですが、何年前か調べたDNSサーバの立て方が、実に簡単になっていたのでご紹介します。

```
$ docker run -d --restart=always \
--publish 53:53/tcp --publish 53:53/udp --publish 10000:10000/tcp \
--volume /srv/docker/bind:/data \
sameersbn/bind:9.11.3-20190706
```

なお、Ubuntuなどではデフォルトでport 53を専有するサービスがあるのでこれをstopしておく必要があるかもしれません。

```
$ sudo systemctl stop systemd-resolved
$ sudo ln -sf /run/systemd/resolve/resolv.conf /etc/resolv.conf
```

これで何かドメインが与えられたときに、解決が自分のDNSを見るか（最初の一回だけキャッシュするために遅くなるが以降は早い）、毎回GoogleかCloudFlareに聞くなど変わってくるので、総合的に自分のローカル環境にdockerなどでいいのでbind9を入れておくと便利です。