

# Crawler Web

---

## Mini-Projet

### Objectifs

Dans ce projet, nous allons implémenter un simple crawler Web ([http://fr.wikipedia.org/wiki/Robot\\_d%27indexation](http://fr.wikipedia.org/wiki/Robot_d%27indexation)), c'est-à-dire un outil capable de parcourir des pages Web en suivant les URLs. C'est typiquement ce que font les moteurs de recherche comme Google. Notre objectif ici est de jouer avec certains des concepts importants que nous avons découverts dans ce MOOC et de pratiquer quelques modules de la librairie standard, mais nous ne chercherons pas la performance parce que ça augmenterait très rapidement la complexité du code et la difficulté du sujet. Cependant, vous constaterez que même si ce crawler n'est pas adapté à crawler des millions de pages, il est parfaitement capable de crawler des milliers de pages et de vous rendre des services (comme identifier les liens morts sur votre site Web).

### Réalisation du crawler Web

Ce projet est découpé en deux niveaux de difficulté. Nous allons commencer par le niveau avancé qui va vous demander d'écrire vous même tout le code en fonction de nos spécifications de haut niveau. Pour le niveau intermédiaire, nous vous fournirons une description plus précise de notre implémentation. À vous de choisir où vous voulez commencer, mais notez que si vous voulez faire le niveau intermédiaire, vous devrez quand même lire le niveau avancé, parce qu'il contient des informations importantes sur le fonctionnement du crawler.

Je vous rappelle qu'une page Web est écrite dans un langage déclaratif qui s'appelle HTML ([http://fr.wikipedia.org/wiki/Hypertext\\_Markup\\_Language](http://fr.wikipedia.org/wiki/Hypertext_Markup_Language)) et que l'on accède à ces pages au travers d'un protocole qui s'appelle HTTP ([http://fr.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](http://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol)). Nous utiliserons dans ce mini projet la librairie standard `urllib2` qui permet d'utiliser le protocole HTTP de manière très simple. Par contre, pour l'interprétation du code HTML, nous ferons tout le traitement à la main. Il existe des librairies pour vous faciliter la tâche (nous en parlerons tout à la fin), mais elles supposent une bonne compréhension des concepts sous-jacents à HTML et XML. Naturellement, vous pouvez les explorer et les essayer dans ce mini projet si vous le souhaitez.

### Niveau avancé

Le but de notre crawler est le suivant. Nous supposons que nous avons un ensemble (set) de pages Web à crawler, c'est-à-dire, un ensemble de pages Web pour lesquelles on va télécharger le code HTML. On commence avec une seule page que l'on appelle la page initiale. Dans la suite cette page sera

`http://www-sop.inria.fr/members/Arnaud.Legout/` (`http://www-sop.inria.fr/members/Arnaud.Legout/`)

On suppose également que l'on restreint le crawl à un sous ensemble de pages Web. Pour cela on définit une liste de pages filtres (`page_filter`) tel que si n'importe quel élément de `page_filter` est contenu dans l'URL de la page que l'on crawl, alors la page passe le filter. Dans la suite, on aura

```
page_filter = www-sop.inria.fr/members/Arnaud.Legout
```

Ensuite notre crawler va :

- prendre un lien (sans ordre particulier) dans l'ensemble des pages Web à crawler
- se connecter à la page correspondant à ce lien, puis
  - enregistrer le code HTTP (`http://fr.wikipedia.org/wiki/Liste_des_codes_HTTP`) associé à cette page (par exemple, 202 lorsque la requête a été traitée correctement et 404 lorsque le lien est mort). On considérera aussi un code -1 lorsque le site Web ne répond pas.
  - récupérer le code HTML de la page
  - si la page passe `page_filter` (typiquement si `filter in page` pour n'importe quel filter dans `page_filter`) extraire toutes les URLs dans le corps de la page Web (c'est-à-dire, après la balise `<body>`) et ajouter à l'ensemble des pages Web à crawler toutes les URLs commençant par `http` ou `https` et toutes les URLs relatives reconstruites commençant par `./` ou `/`
    - par exemple, pour `./ma_page.html` et le site `http://mon_site.fr/rep1/` on ajoute l'url `http://mon_site.fr/rep1/ma_page.html`
    - par exemple, pour `/ma_page.html` et le site `http://mon_site.fr/rep1/` on ajoute l'url `http://mon_site.fr/ma_page.html` (sans `rep1`)
- on recommence au premier point jusqu'à ce que l'ensemble des pages Web à crawler soit vide.

Pour simplifier, on va manipuler le crawler comme un itérable, à chaque appel de `next()` on fait avancer le crawler d'une page (dans l'ensemble des pages à crawler) et on obtient un objet qui contient le code HTTP pour la page, l'URL de la page, et la liste de toutes les URLs contenues dans la page (extraites comme expliqué ci-dessus).

Le but de ce mini projet est d'utiliser le crawler pour identifier tous les liens défectueux (c'est-à-dire ceux pour lesquels HTTP ne retourne pas un code entre 200 et 299) pour le site `http://www-sop.inria.fr/members/Arnaud.Legout/`.

Voici le résultat de l'exécution du crawler avec comme page initiale `http://www-sop.inria.fr/members/Arnaud.Legout/` et comme `page_filter` [`www-sop.inria.fr/members/Arnaud.Legout`]. Donc le crawler va uniquement tester les liens qui sont dans le site `http://www-sop.inria.fr/members/Arnaud.Legout/`



Page contenant des liens defectueux :

**`http://www-`**

## **sop.inria.fr/members/Arnaud.Legout/Projects/p2p\_cd.html**

CODE HTTP 404 <http://dx.doi.org/10.1016/j.comnet.2010.09.014>  
(<http://dx.doi.org/10.1016/j.comnet.2010.09.014>)

 <http://www.cs.ucla.edu/~nikitas/>

=====

Page contenant des liens defecteux :

## **<http://www-sop.inria.fr/members/Arnaud.Legout/publications.html>**

CODE HTTP 404


 <http://dx.doi.org/10.1016/j.comnet.2010.09.014>

=====

Page contenant des liens defecteux :

## **<http://www-sop.inria.fr/members/Arnaud.Legout/Projects/bluebear.html>**

CODE HTTP 303 <http://bits.blogs.nytimes.com/2011/11/29/skype-can-expose-your-location-researchers-say/> (<http://bits.blogs.nytimes.com/2011/11/29/skype-can-expose-your-location-researchers-say/>)  
CODE HTTP 403 [https://threatpost.com/en\\_us/blogs/attacking-and-defending-tor-network-032911](https://threatpost.com/en_us/blogs/attacking-and-defending-tor-network-032911) ([https://threatpost.com/en\\_us/blogs/attacking-and-defending-tor-network-032911](https://threatpost.com/en_us/blogs/attacking-and-defending-tor-network-032911))  
<http://www.pcinpact.com/actu/news/66544-skype-bittorrent-etude-scientifiques-faille.htm>  
(<http://www.pcinpact.com/actu/news/66544-skype-bittorrent-etude-scientifiques-faille.htm>) CODE HTTP 404

 <http://www.zataz.com/news/21651/faille--vulnerability-skype.html>

=====

Page contenant des liens defecteux :

## **<http://www-sop.inria.fr/members/Arnaud.Legout/>**

CODE HTTP -1

 <http://www.castify.net>

=====

Page contenant des liens defectueux :

## http://www-sop.inria.fr/members/Arnaud.Legout/index.html

CODE HTTP -1

```
⌘ http://www.castify.net
```

=====

### Niveau intermédiaire

Tout notre programme peut-être écrit dans un même module `webcrawler`. Nous avons dans ce module deux classes. Voici l'aide correspondant à ces deux classes

#### Classe `HTMLPage`

```
⌘ Help on class HTMLPage in module webcrawler:
```

`class HTMLPage(builtins.object)` | represente une page HTML. | | L'objet a 4 attributs: | `-url`: l'URL qui correspond a la page Web | `-htmlit`: un *iterateur qui parcourt le code HTML, une ligne | a la fois* | `-urls`: la liste de toutes les URLs contenues dans la page | `-http_code`: le code retourne par le protocol HTTP lors de | l'acces a la page | `http_code=0` signifie une erreur dans l'URL, | `http_code=-1` signifie que le site de repond pas | `*http_code=-2` signifie une exception en accedant | a l'URL | | *Methods defined here:* | | `__init__(self, url)` | Constructeur de la classe. Le constructeur prend comme | argument une URL et construit un objet `HTMLPage` en definissant | les 4 attributs `url`, `_html_it`, `urls`, `http_code` | | `extract_urls_from_page(self)` | Construit la liste de toutes les URLs contenues dans le corps de | la page HTML en parcourant l'iterateur retourne par | `page_fetcher()` | | On identifie une URL parce qu'elle est precedee de `href=` et | dans le corps (body) de la page. Le parsing que l'on implement | est imparfait, mais un vrai parsing intelligent demanderait | une analyse syntaxique trop complexe pour nos besoins. | | Plus en details, notre parsing consiste a chercher dans le | corps de la page (body): | | -les urls contenues dans le champ `href` (essentiellement on | cherche le tag `'href='` et on extrait ce qui est entre | guillemets ou apostrophes) | | -on ne garde ensuite que les urls qui commencent par `http` ou | `https` et | | *les urls qui commencent par ./ auxquelles on ajoute | devant (a la place du point) l'Url de la page d'origine | (self.url) exemple : pour './ma\_page.html' et self.url = | http://mon\_site.fr/rep1/ (http://mon\_site.fr/rep1/) on obtient l'url | http://mon\_site.fr/rep1/ma\_page.html (http://mon\_site.fr/rep1/ma\_page.html) | | les urls qui commencent par /ma\_page.html auxquelles on | ajoute devant uniquement le hostname de la page d'origine | (self.url) exemple : pour '/ma\_page.html' et self.url = | http://mon\_site.fr/rep1/ (http://mon\_site.fr/rep1/) on obtient l'url | http://mon\_site.fr/ma\_page.html (http://mon\_site.fr/ma\_page.html) | | Cette methode retourne la liste des URLs contenues dans la | page. | | `page_fetcher(self, url)` | accede a l'URL et retourne un objet qui permet de parcourir le | code HTML (voir la documentation de `urllib2.urlopen`) ou une | liste vide en cas d'erreur. |*

#### Classe `Crawler`



Help on **class** Crawler in module webcrawler:

class Crawler(**builtin**.object) | Cette classe permet de creer l'objet qui va gerer le crawl. Cet objet est iterable et l'iterateur va, a chaque tour, retourner un | nouvel objet HTMLPage. | | L'instance du crawler va avoir comme principaux attributs | *l'ensemble des pages a crawler* `pages_to_be_crawled` | l'ensemble des pages deja cawles `pagescrawled` | *un dictionnaire qui a chaque URL fait correspondre la liste de* | *toutes les pages qui ont reference cette URL lors du crawl* | `pagestobe_crawled_dict` | | *Methods defined here:* | | `__init__(self, seed_url, max_crawled_pages=10000000000L, page_filter=None)` | Constructeur du crawler | | Le constructeur prend comme arguments | `-seed_url`: l'URL de la page a partir de laquelle on demarre le crawl | `-max_crawled_pages`: le nombre maximum de pages que l'on va crawler | (10\*10 par default) | `-pagefilter`: la liste des pages sur lesquels le crawler | doit rester (pas de filtre par default). Typiquement, une URL | passe le filtre si n'importe lequel des elements de `pagefilter` | est contenu dans l'URL | | `__iter__(self)` | Cette methode est implementee comme une fonction generatrice. A | chaque appel de `next()` sur l'iterateur, on obtient un nouvel | objet HTMLPage qui correspond a une URL qui etait dans | l'ensemble des URLs a crawler. | | On ne donne aucune garantie sur l'ordre de parcours des URLs | | `__repr__(self)` | permet d'afficher simplement des informations sur l'etat | courant du crawl. | | retourne une chaine de caracteres donnant: | -le nombre de pages et domaines deja cawle | -le nombre de pages encore a crawler | -la duree du dernier crawl | | `update_pages_to_be_crawled(self, page)` | Prend un objet HTMLpage comme argument et trouve toutes les | URLs presente dans la page HTML correspondante. Cette methode | met a jour le dictionnaire `pages_to_be_crawled_dict` et | l'ensemble `pages_to_be_crawled`. On ne met pas a jour le | dictionnaire et le set si l'URL correspondant a l'objet | HTMLpage n'est pas dans la liste de pages acceptees dans | `self.page_filter`. |

### Fonctions utilitaires

On a également deux fonctions, qui sont utilisées par les classes, dont voici l'aide.



Help on **function** `extract_domains_from_url` in module webcrawler:

`extract_domains_from_url(url)` Extrait un domaine d'une URL



Retourne le tuple T qui contient

T[0]: domaine avec le bon protocol (`http://domain` or `https://domain`)  
T[1]: domaine sans le protocol (sans `http://` or `https://`)

Help on **function** `is_html_page` in module webcrawler:

`is_html_page(url)` simple heuristique pour tester si une page est ecrite en HTML. Il y a des cas mal identifiés par cette heuristique, mais elle est suffisante pour nos besoins. Par exemple: `http://inria.fr` (`http://inria.fr`) sera identifié comme non html de meme que toutes les pages qui utilisent des points dans le nom d'un repertoire.

### Le mot de la fin

Nous avons à de nombreuses reprises évoqué la puissance de la librairie standard, mais aussi des librairies tierces. En particulier, nous avons insisté sur le fait qu'au démarrage d'un projet, il vaut mieux commencer par chercher si une librairie Python ne fait pas déjà tout ou partie de

ce que vous voulez faire.

Il existe une librairie Python très puissante qui permet justement de faire des crawlers : il s'agit de Scrapy (<http://scrapy.org/>) . Maintenant que vous avez compris les bases d'un crawler Web, vous pourrez tirer pleinement bénéfice de Scrapy.

Il existe également une librairie pour parser du code HTML, c'est BeautifulSoup (<http://www.crummy.com/software/BeautifulSoup/>) .