Projet Ding

1. Introduction

Le projet Ding consiste à implémenter un indexeur pour le web. A terme, celui-ci pourra comporter les éléments suivants :

* un crawler : logiciel permettant de récupérer le contenu de pages HTML sur le web, et de suivre des liens pour explorer incrémentalement le réseau

1. Expression régulière

Les expressions régulières, ou *regex*, sont des “patterns” qui permettent de rechercher des mots ayant certaines propriétés.

Pour introduire ce concept, j’ai été amené à créer 2 fonctions : def get\_word et def search. Celles-ci ont pour ambition d’isoler des mots pour ensuite y appliquer un critère de recherche.

* def get\_word : Prend en paramètre un fichier puis, compte le nombre d’occurrence de chaque mot qu’il contient et renvoie les informations sous forme de dictionnaire.
* def search : Prend en paramètre un dictionnaire et comme second argument une chaine de caractère étant le mot que l’on veut rechercher

Cette chaîne peut simplement se composer d'un mot unique, dont il faudra alors chercher simplement l'occurrence, mais peut contenir le caractère spécial '\*', qui représentera n'importe quelle suite de caractères. Ainsi, par exemple, la chaîne \*test correspond à tous les mots qui se terminent par test.

Ma fonction se décompose de la manière suivante :

if search\_str[0] == "\*":

search\_str = search\_str.replace("\*","(.)\*")

elif search\_str[-1] == "\*":

search\_str = search\_str.replace("\*","(.)\*")

elif "\*" in search\_str: search\_str = search\_str.replace("\*","(.)\*")+ "$"

key\_list = [(k,val) for (k, val) in d.items() if re.match(search\_str,k)]

Sortie :

>>> r = search(d, "mo\*")

>>> sorted([c[0] for c in r])

['mon', 'monsieur', 'montrer', 'mots']

Expression régulière : mo(.)\*

>>> r = search(d, "\*age")

>>> sorted([c[0] for c in r])

['fromage', 'langage', 'plumage', 'ramage']

Expression régulière :  (.)\*age

>>> r = search(d, "\*on\*")

>>> sorted([c[0] for c in r])

['bon', 'bonjour', 'confus', 'honteux', 'leçon', 'mon', 'monsieur']

Expression régulière: (.)\*on(.)\*

>>> r = search(d, "p\*e")

>>> sorted([c[0] for c in r])

['plumage', 'proie']

Expression régulière : p(.)\*e$

Le symbole \* nombre de caractères attendus 0,1 ou plus

. Le point correspond à n'importe quel caractère.

^ Indique un commencement de segment mais signifie aussi "contraire de"

$ Fin de segment

1. Crawler

Il s’agit de récupérer la page HTML d’un site cible afin de l’analyser. Fonctionnant de la manière suivante :

* isoler les mots et en calculer le nombre d'occurrences dans la page
* isoler les hyperliens et en calculer le nombre d'occurrences

**Epurer la page HTML**

Il faut premièrement épurer la page HTML , c’est-à-dire retirer les ancrages. Pour ce faire, il est nécessaire de se référer aux expressions régulières.

Ex r “<[^>]\*> ” va nous permettre de retirer tous les ancrages de la page.

**Les différents ensembles**

**m[oai]ts** mots, mats et mits

**m[^oai]ts** mets

Les ensembles de caractères permettent aussi d'exclure des caractères grâce à l'accent circonflexe ^.

**Isoler les hyperliens (href=)**

Il nous faut maintenant isoler les hyperliens pour pouvoir les stocker dans un dictionnaire à l’aide de l’expression régulière suivante :

m = re.findall("href=[\"](.\*?)[\"]", html)

La variable m va contenir tous les hyperliens de ma page HTML.

En utilisant findall à la place de search cela va créer une liste qui va reprendre tous les éléments qui matcheront avec le fait que la chaine de caractère commencant par href=" ….."

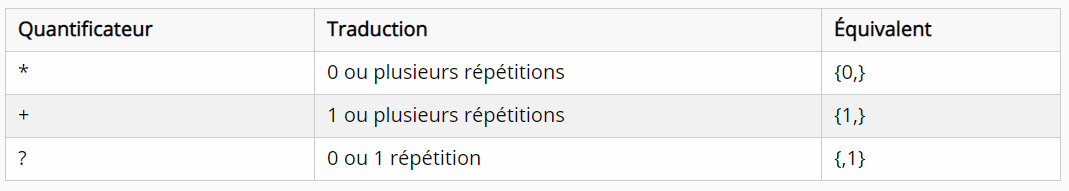
La barre oblique inverse (\) dans une expression régulière indique que le caractère qui la suit est un caractère spécial. Dans notre cas les " "

>>> Tout ce qui débutera par href=" et finira par "

Mise en situation

<a href="<http://127.0.0.1:8080/empty.txt>">empty file</a>Cette leçon vaut   
bien un fromage, sans doute. "  
  
<a href="<http://127.0.0.1:8080/notfound.txt>">notfound file</a>  
Le Renard s'en saisit, et dit : "Mon bon Monsieur,

==> lien\_href = re.findall("href=[\"](.\*)[\"]", html)  
  
[http://127.0.0.1:8080/empty.txt">empty](http://127.0.0.1:8080/empty.txt%22%3eempty) file</a>Cette leçon vaut bien un   
fromage, sans doute.  
  
[http://127.0.0.1:8080/notfound.txt">notfound](http://127.0.0.1:8080/notfound.txt%22%3enotfound) file</a>Le Renard s'en   
saisit, et dit :  
  
==>lien\_href = re.findall("href=[\"](.\*?)[\"]", html)  
  
<http://127.0.0.1:8080/empty.txt>  
  
<http://127.0.0.1:8080/notfound.txt>



Cela nous permet de comprendre que le « ? » à la première répétition

1. Paramètre optionnel
2. Optimisation

La complexité en temps est une mesure du temps utilisé par un algorithme. Il est donc nécessaire d’optimiser son code par soucis d’efficacité. La complexité permet de quantifier la relation entre les conditions de départ et le temps effectué par l'algorithme.

Pour représenter cette approximation on utilise une notation spécifique, la notation O.

Durant la création de ma fonction def get\_word, j’ai été amené à garnir un dictionnaire. La façon d’y accéder et de le remplir peut se faire de différentes manières et par conséquent de manière plus ou moins optimale.

Recherche dans le dictionnaire :

# faster

for word in words:

try:

wdict[word] += 1

except KeyError:

wdict[word] = 1

# nicer

get = wdict.get

for word in words:

wdict[word] = get(word, 0) + 1

Eviter de boucler si possible :

if url is not None:

if url in dic\_url\_db:

return dic\_url\_db.get(url)

Il est préférable de regarder en premier lieux si la variable est remplie afin de ne pas boucler pour rien

1. Récursivité
2. Doctest
3. API

Def web\_index( url, minl)

La fonction va prendre en argument une url, à partir de cette url nos allons extraire sa page HTML et l’assainir.

Assainissement en 2 étapes :

Un dictionnaire « words » contenant le nombre d’occurrence de chaque mot.

Un dictionnaire « links » contenant le nombre d’occurrence de chaque href.

Nous allons créer un backend de stockage, pour le biais de variables globales.

Un dictionnaire global qui regroupera les URL et les liens qu’elles contiennent :



XXX faisant référence au timestamp

On pourra les enregistrer avec la fonction store\_ruls( ), et on pourra accéder à ce dictionnaire global par la fonction get\_urls.

On fera la même chose pour les mots et leurs occurrences avec la fonction store\_data () et get\_data

Contextualisation :

url = "<http://127.0.0.1:8080/empty.txt>"

tt = store\_urls(url,{ "[http://a.com](http://a.comt)" : 1 }

d = get\_urls(url)

d >>> (1613123755.052092, {'<http://a.com>': 1})

Le dictionnaire global contiendra les informations suivantes :   
{'<http://127.0.0.1:8080/empty.txt>': (1613123755.052092, {'<http://a.com>':   
1})}

Def store­\_urls(url1, urls) :

Prend comme argument URL1 (le site target) et prends URLS( le dictionnaire d’URL(href) lié au site target). On implémente le dictionnaire d’URLS dans le dictionnaire global.

En résumé on store(stock) les urls dans un global dico.

La fonction get\_urls permet de regarder dans le global dico proprement

Si la fonction store\_urls, reçoit comme un argument => url, None

La fonction devra alors nettoyer les informations lié a cette url, on faisant un pop nous allons le retirer du dictionnaire et ainsi supprimer la key et sa value et renvoyer 0.

Si la fonction get\_urls ne reçoit aucun arguments alors la fonction devra renvoyer toutes les clefs du dictionnaire globale c’est-à-dire afficher les sites qui ont été analysés

Même procédé pour get\_data, store\_data

1. Stockage persistant

Dans un premier temps, nous avons référencé les 2 dictionnaires par des variables global. Nous allons maintenant rendre le stockage persistant en utilisant pickels :

Db\_urls.pick

Db\_words.pick

Les fonctions de l’API devront consulter et enregistrer leurs données dans ces fichiers. Chaque enregistrement et chaque consultation doit alors se faire vers et à partir des fichiers pickel.txt

Fonctionnement de pickle (last in, first out) :

>>> import pickle  
>>> a = "chaine A"  
>>> b = "chaine B"  
>>> with open('/tmp/test.pkl', 'wb') as f:  
... pickle.dump(a, f)  
... pickle.dump(b, f)  
...  
>>> with open('/tmp/test.pkl', 'rb') as f:  
... print(pickle.load(f))  
... print(pickle.load(f))  
...  
chaine A  
chaine B  
>>> with open('/tmp/test.pkl', 'rb') as f:  
... print(pickle.load(f))  
... print(pickle.load(f))  
...  
chaine A  
chaine B

/ ! \

Tant que le fichier est ouvert lorsque l’on dump ça s’empile, mais lorsque l’on dump à nouveau dans le même fichier après qu’il ait été fermé au préalable et que l’on le réouvre , on écrase tout.

Le dictionnaire global créer dans notre API nous servira d’intermédiaire (de buffer) pour lire et écrire dans le fichier pickle

Nous allons écrire une nouvelle fonction « store\_urls\_pickle ». Cette nouvelle fonction appelle l’ancienne pour stocker les données dans le dictionnaire global, puis enregistre le dictionnaire global dans le fichier pickle.

Pour les fonctions de lecture comme get\_urls, le fonctionnement est inverse : la nouvelle fonction « get\_urls\_pickle » va lire le fichier pickle, charger le dictionnaire global, puis appeller l'ancienne fonction

1. Décorateur

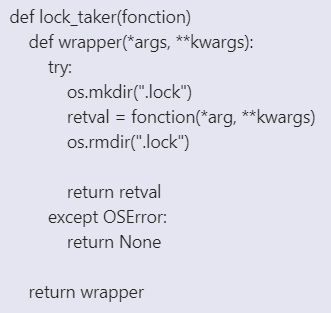
Tout d’abord, les fonctions sont des valeurs que l’on peut affecter ou passer en argument. On peut remplacer une fonction par une autre :

def divide(x,y):  
     return x/y  
  
def add(x,y):  
     return x+y  
  
divide = add  
x = divide(3,5)  
  
print(x) >>> 8

Pour introduire le concept de décorateur, nous avons été confrontés à la création d’un mécanisme de verrou. Celui-ci nous permettant de gérer les accès concurrents au sein du programme. Il s'agit d'un simple verrou exclusif basé sur l'utilisation de la présence d'un répertoire prédéterminé comme "sémaphore".

On comprend donc que cette fonction doit être nécessairement affecté à chaque fonction de notre API.

Le décorateur va permettre d’incorporer cette nouvelle fonction sans devoir la réécrire en dur sur chaque fonction de l’API.



Def loc\_taker va faire référence au nom du décorateur

Wrapper va renvoyer retval lorsqu’il arrive à créer le répertoire. En d’autres termes, la fonction de base qui a été enrobée va être lancée si mkdir fonctionne et le résultat va être stocké dans retval pour ensuit être return

Lock\_taker va renvoye Wrapper

**Fonctionement théorique**

def divide(x,y):  
     return x/y

def ajout\_addition(function\_divide):  
     def wrapper(arg1, arg2):  
         retval = function\_divide(arg1, arg2)  
         add = arg1 + arg2  
         return retval, add  
     return wrapper

divide = ajout\_addition(divide)

Lorsque l’on décore une fonction, celle-ci est enrobée, détruisant ainsi ses informations. En effet divide contient maintenant WRAPPER et non la fonction initiale.

Pour pouvoir supporter les arguments, nous avons besoin de la fonction wrapper. Celle-ci accepte les arguments. (wrapper = fonction de base + décoration). Renvoie la même chose avec des décorations en plus.

**Transformation de la fonction « store\_urls\_pickle » en décorateur**

|  |
| --- |
| Fonction |
| def store\_urls\_pickle(url, url\_dict):    store\_urls(url, url\_dict)    with open(db\_u,'wb') as f:  pickle.dump(dic\_url\_db, f |

|  |
| --- |
| Décorateur |
| def store\_urls\_pickle(fonction):  def wrapper(\*arg, \*\*kwargs):    fonction(\*arg, \*\*kwargs)  with open(db\_u,'wb') as f:  pickle.dump(dic\_url\_db, f)  return wrapper |

@store\_urls\_pickle

def store\_urls(url, url\_dict):

global dic\_url\_db

if url\_dict:

stamp = mtime()

dic\_url\_db[url] = stamp , url\_dict

return stamp

else:

if url in dic\_url\_db:

dic\_url\_db.pop(url)

return 0

**Transformation de la fonction « store\_urls\_pickle » en décorateur à argument**

De cette manière notre décorateur devient plus fexible, ainsi celui-ci peut gérer la fonction store\_data(même fonction mais qui gère des mots)

def store\_universal(db, dic):

def store\_urls\_pickle(fonction):

def wrapper(\*arg, \*\*kwargs):

fonction(\*arg, \*\*kwargs)

with open(db,'wb') as f:

pickle.dump(dic, f)

return wrapper

return store\_urls\_pickle

@store\_universal(db\_u,dic\_url\_db)

def store\_urls(url, url\_dict):

global dic\_url\_db

if url\_dict:

stamp = mtime()

dic\_url\_db[url] = stamp , url\_dict

return stamp

else:

if url in dic\_url\_db:

dic\_url\_db.pop(url)

return 0

Nous passons en paramètre quelle DB utiliser et quel dictionnaire

* db\_u, dic\_url\_db
* db\_w, dic\_word\_db

**Transformation de la fonction « get\_urls\_pickle » en décorateur**

def get\_urls\_pickle(url=None):

with open(db\_u,'rb') as f:

dic\_url\_db = pickle.load(f)

return get\_urls(url=None)

ERREUR return

def store\_universal(db, dic):

def store\_urls\_pickle(fonction):

def wrapper(\*arg, \*\*kwargs):

retval = fonction(\*arg, \*\*kwargs)

with open(db,'wb') as f:

pickle.dump(dic, f)

return retval

return wrapper

return store\_urls\_pickle

@store\_universal(db\_u,dic\_url\_db)

def store\_urls(url, url\_dict):

global dic\_url\_db

if url\_dict:

stamp = mtime()

dic\_url\_db[url] = stamp , url\_dict

return stamp

else:

if url in dic\_url\_db:

dic\_url\_db.pop(url)

return 0

Si je ne return pas retval je ne pourrais pas catch le return 0 ou return stamp

REFAIRE

1)

Renvoie une liste mais je dois renvoyer un dictionnaire ;

result = dict(sorted(result.items(),key=lambda k: k[0].translate(trans)))

result.sort(key=lambda k: k[0].translate(trans))

>>> trans = str.maketrans({"œ":"oe", "æ":"ae", "é":"e",   
"è":"e","ê":"e"}) #etc.  
 >>> l = ["dummy", "alors", "œuf", "advanced", "ætéria", "être", "étirer"]  
 >>> l.sort(key=lambda k: k.lower().translate(trans))  
 >>> l  
['advanced', 'ætéria', 'alors', 'dummy', 'étirer', 'être', 'œuf']

2)

1. Utilisation de python

Il est important d’exploiter pleinement les possibilités qu’offre les structures de données : Liste, tuple, dictionnaires,

La possibilité qu’offre le dictionnaires ne sont pas du tout exploité :

Fonction get

def get\_urls(url=None):

if url is not None:

if url in dic\_url\_db:

return dic\_url\_db.get(url)

else:

return ()

else:

return dic\_url\_db.keys()

def get\_urls(url=None):

if url is not None:

return dic\_url\_db.get(url,())

else:

return dic\_url\_db.keys()

Fonction store

def store\_data(url, word\_dict):

global dic\_word\_db

if word\_dict:

stamp = mtime()

dic\_word\_db[url] = stamp , word\_dict

return stamp

else:

if url in dic\_word\_db:

dic\_word\_db.pop(url)

return 0

def store\_data(url, word\_dict):

global dic\_word\_db

if word\_dict:

stamp = mtime()

dic\_word\_db[url] = stamp , word\_dict

return stamp

else:

dic\_word\_db.pop(url,0)

return 0

NB

Le timestamp (unix) désigne le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit UTC précise

Decorateur parti pickel

Recursion

Pytest pour flask

Unixtest

Consulter un attribut p.fname = fname

Attribuer un attribut

Many to many ,argument Sqlobject

DB en mémoire

\*y positionnel