Introduction au web scraping avec R

Collecte de données automatisée en ligne

Vestin Hategekimana

07.11.2022

Table des matières I

Présentation

moodle

Classe: TO2022

Qui suis-je?

Vestin Hategekimana

- Assistant-doctorant en démographie (IDESO)
  - Institut de socioéconomie et de démographie (UNIGE)
- Migration et mobilité en Suisse en temps de crise
- Passionné des sciences des données (computational social sciences)

WeData

WeData

"Des stats et du code!"

Groupe étudiant ayant une passion pour le code et les statistiques: cours et contenu!

- Notre site: https://wedata.ch/
- Notre chaîne YouTube: WeData
- Instagram: @wedata_unige

Questions d'introducition

Lien votamatic

Code: XRTN

Pour ce cours

Note

Pour le bon déroulement du cours, sachez que:

1. Je suis un amateur passionné
2. C'est mon premier cours à l'université
3. C'est la première fois que j'enseigne le sujet
4. Ce n'est pas un cours formel
5. Vous pouvez partir à n'importe quel moment
6. Vous pouvez m'interrompre si vous avez une question

Introduction au web scraping

Qu'est-ce que le web scraping?

Qu'est-ce que le web scraping?

Note

Web scraping: Processus de collecte de données d'une page web. Le but est de récolter, nettoyer puis de transformer les données dans un format réutilisable.
Le web scraping a plusieurs utilisations:

- Création d'API
- Traçage d'information (business ...)
- Recherche

Comment faire du web scraping?

Les étapes:

1. Choisir un outil (langage de programmation, application)
2. Choisir un site web et l'analyser/l'auditer
3. Planifier la collecte
4. Collecter les données

Le web scraping dans R

Pour pouvoir collecter des données dans R, nous pouvons utiliser des fonctions de base pour quelques tâches, mais il existe des packages qui permettent d'améliorer le processus.

> ***Package**: Ensemble de fonctions, de code et/ou de données regroupés dans une extension.*

Le package le plus populaire pour le web scraping dans R est {rvest}. Il a une structure simple et intéressante de gérer les informations d'une page internet. Ce package s'inscrit dans ce qu'on appel le {tidyverse}.

Tidyverse

Dans le cadre de ce cours, nous allons nous baser sur les packages du {tidyverse}. C'est un ensemble de packages partageant une philosophie commune "tidyformat" qui sont dédié au nettoyage et a la manipulation de données.

Sources: tidyverse

> ***Tidyformat**: un data frame (tableau de données) avec en ligne des individus et en colonne des variables.*

{rvest}

Nous allons principalement utiliser le package {rvest} qui a été spécialement fait pour le web scraping. Mais nous nous aiderons d'autres packages du {tidyverse} plus spécifique.

Il faut donc tout charger ensemble.

Installation des packages

Avec cette fonction, on installe tous les packages de tidyverse, les 8 de base plus des bonus (ex. rvest, lubridate, dtplyr, etc.).

```
install.packages("tidyverse")
```

Préférences

Les projets dans R

Les avantages d'un projet:

- Permet d'avoir des dossiers séparés avec leur propre dossier de travail, historique et source pour les documents.
- Contribue aussi à la reproductibilité de votre travail

Tutoriel

Mise en place du projet

1. Commencer un projet sur R
2. Créer un dossier pour les données
3. Ouvrir un script R (ou Rmarkdown)

Bien documenter son code et garder une certaine hygiène permet de se faire une ressource sûre pour le future!

Quelques règles d'hygiène

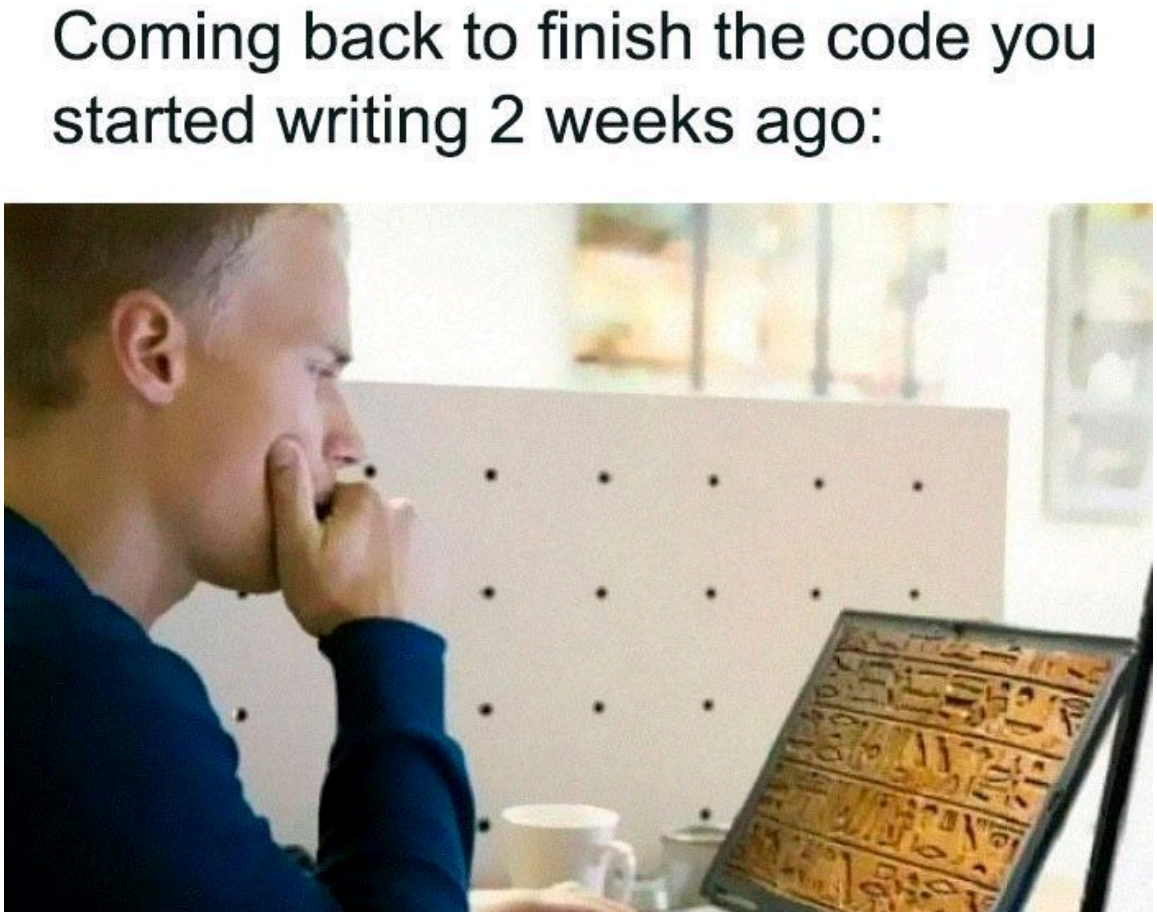
Voici quelques idées pour garder un code propre:

1. Garder un code chronologiquement correct
2. Avoir des séctions spécifiques voire d'autres scripts spécifiques
3. Dans le cadre d'un code "ressources" commenter suffisement ce qui se fait.
4. Avoir des noms de variable explicites
5. Espacer son code pour la lisibilité
6. Avoir un dossier de travail en ordre et standard (projet)

On doit pouvoir lancer tout le code en une fois sans problème. C'est embêtant au début mais c'est bénéfique!

Bénéfices

1. Code partageable, donc reproductible!
2. Facililter de se relir donc ressource!



Exercice importer des données

Importer des fichiers

Sur internet, il existe une série de données gratuites mises à dispositions par des particuliers. Notamment sur github:

Données

Explorons tout ça!

Importer des csvs

Si les données sont au format csv, nous pouvons simplement les importer avec les fonctions `read.csv()` ou `read_csv()`.

```
read.csv(
  "https://raw.githubusercontent.com/datasets/corruption-pe
  )
```

```
library(readr) # Dans {tidyverse}
read_csv("https://raw.githubusercontent.com/datasets/corrup
```

```
# Même chose mais enregistré dans un objet manipulable
cpi <- read_csv("https://raw.githubusercontent.com/datasets
```

Sauvegarder des données

Une fois que les données sont importées, il est conseillé de les sauvegarder avant et/ou après leur nettoyage. Dans tous les cas il faut les sauvegarder.

R vous permet de sauvegarder dans plusieurs formats. Pour pouvoir utiliser les différentes fonctions qui suivent, il faut importer ces packages supplémentaires:

```
install.packages("readr")
install.packages("haven")
install.packages("rio")
```

Export: R

On peut sauvegarder autant d'éléments qu'on veut dans des données R (.rda, .rdata)

```
save(cpi, file = "cpi.rda")
```

Export: csv

Base

```
# Séparateur en virgule
write.csv(cpi, file = "cpi.csv")
# Séparateur en point-virgule
write.csv2(cpi, file = "cpi.csv")
```

{readr}

```
# Séparateur en virgule
write_csv(cpi, file = "cpi.csv")
# Séparateur en point-virgule
write_csv2(cpi, file = "cpi.csv")
```

Export: SPSS, SAS & STATA

{haven}

```r
library(haven)

# SPSS
write_sav(cpi, path = "cpi.sav")

# SAS
write_sas(cpi, path = "cpi.sas7bdat")

# STATA
write_dta(cpi, path = "cpi.dta")
```

Export: multiple format

Le package {rio} simplifie la vie pour l'export et l'import de données en gérant tous les formats de données et d'extension avec seulement deux fonctions `import()` et `export()`:

```
library(rio)

# Exemple au format excel
export(cpi, file = "cpi.xlsx")
```

formats: csv, psv, tsv, SAS, SPSS, STATA, Excel, R, JSON, minitab, Eview, Matlab, html, etc.

{rio}

Exercice

Importez des données csv de votre choix et sauvegardez-les dans le format désiré.

Données

Les tableaux en ligne

Il est possible de collecter des données de tableau en ligne. Pour cela nous devons commencer à utiliser le package {rvest}:

```
library(rvest)
```

Nous allons collecter les données du tableau sur l'indice de corruption (corruption index) du site web "tranding economics":

https://tradingeconomics.com/country-list/corruption-index

Les tableaux en ligne

Premièrement nous devons établir une connexion au site avec la fonction `read_html()`:

```
url <- "https://tradingeconomics.com/country-list/corruptio

page <- read_html(url)
```

Nous obtenons un document XML que nous pouvons analyser:

```
page
## {html_document}
## <html>
## [1] <head id="ctl00_Head1">\n<meta http-equiv="Content-T
## [2] <body>\r\n    <script>IsDarkMode=false;</script><fo
```

Les tableaux en ligne

Nous cherchons premièrement les éléments “table” (les tableaux) avec la fonction `html_elements()` ou `html_nodes()`. Mais nous allons utiliser une “pipe function” %>% (ctrl+Maj+M ou cmd+Maj+M) pour suivre dans le code:

```
page %>%
  html_elements("table")
## {xml_nodeset (1)}
## [1] <table class="table table-hover table-heatmap" data
```

Nous avons un résultat (1) donc nous avons réussi à trouver une table!

La fonction “pipe”

La fonction “pipe” (%>%) est une fonction importante du {tidyverse}. Elle permet de transférer les tâches entre plusieurs fonctions.

De manière simplifiée, cette fonction dit à la fonction qui la suit

> *“Prend le résultat de la fonction ou de l’objet à ma gauche et utilise le comme ton premier argument.”*

Équivalence

Le code de droite et de gauche donne le même résultat en passant par: les mêmes étapes. Ici encore c'est une question de préférence.

```r
V1 <- 1:10
V1 <- scale(V1)
V1 <- summary(V1)
```

```r
V1 <-
  1:10 %>%
  scale() %>%
  summary()
```

Les tableaux en ligne (suite)

Maintenant que nous avons trouvé une table dans l'objet "page", nous pouvons la formater pour la transformer en tableau de données (data frame) avec la fonction `html_table()`:

```
page %>%
  html_elements("table") %>%
  html_table()
## [[1]]
## # A tibble: 180 x 5
##    Country      Last Previous Reference Unit
##    <chr>       <int>    <int> <chr>     <chr>
##  1 Denmark        88       88 Dec/21    Points
##  2 Finland        88       85 Dec/21    Points
##  3 New Zealand    88       88 Dec/21    Points
##  4 Norway         85       84 Dec/21    Points
##  5 Singapore      85       85 Dec/21    Points
```

Les tableaux en ligne (suite)

Nous pouvons enregistrer notre tableau de donnée dans un objet pour l'utiliser plus tard.

```
tableau <-
  page %>%
  html_elements("table") %>%
  html_table()
```

Le grand avantage de cette méthode est la rapidité avec laquelle nous pouvons obtenir des données. On évite ainsi de devoir retranscrire les données à la main dans un tableau Excel par exemple (même s'il y a une méthode de collecte similaire sur Excel).

Bonus

Si vous voulez faire le travail manuellement, vous pouvez aussi utiliser le package {datapasta}:

{datapasta}

Démonstration

```
install.packages("datapasta")
```

```
library(datapasta)
```

Avantages:

- Marche sur plusieurs formats (html, excel, etc.)
- Pas de code
- Pratique pour des petits tests

Bonus

Désavantages:

- Pas de trace
- Pas automatisable

Plusieurs tableaux

Lorsqu'il y a plusieurs tableaux dans une page, nous obtenons une liste de tableaux. Cette liste forme un seul objet.

Lien d'exemple

Pour choisir une seule table, nous devons préciser sa position.

```
url <- "https://en.wikipedia.org/wiki/List_of_countries_by_

page <- read_html(url)

page %>%
  html_elements("table") %>%
  html_table() %>%
  .[[3]]
```

Exercice (Wikipédia)

Temps: ~5 minutes

Choisir un ou plusieurs tableaux, le(s) collecter et le(s) enregistrer.

Tableaux Wikipédia

Obtenir du texte

Nous pouvons aussi collecter des données textuelles d'une page. Il nous suffit de remplacer "table" par "h1, h2, h3, p" et remplacer `html_table()` par `html_text2()`:

```
url <- "https://fr.wikipedia.org/wiki/Web_scraping"

page <- read_html(url)

wiki <-
  page %>%
  html_elements("h1, h2, h3, p") %>%
  html_text2()
```

Cette fois-ci nous obtenons une chaîne de caractères.

Export: texte

Base

```
writeLines(wiki, con = "wiki.txt")
```

{readr}

```
write_lines(wiki, file = "wiki.txt")
```

Exercice

Temps: ~5 minutes

Collecter les données textuelles d'une page web de votre choix et sauvegarder les données en texte (fichier TXT).

Par exemple, journal en ligne, blog, autre site, etc.

Image et autre type de fichier

Nous pouvons aussi télécharger des images ou d'autres types de fichiers (pdf, excel, etc.). Cette fois-ci les images ne passent pas par R, mais sont directement enregistrées en tant que fichier.

La fonction `download.file()` qui est de base dans R permet de télécharger plusieurs types de données:

```
url <- "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd

download.file(url,
              destfile = "castor.jpg",
              mode = "wb")
```

Exemple fichier Excel

Même chose avec un fichier Excel de l'Office Fédérale de la Statistique (OFS):

```
url <- "https://dam-api.bfs.admin.ch/hub/api/dam/assets/236

download.file(url,
              destfile = "ofs.xlsx",
              mode = "wb")
```

Téléchargement export (1)

Note

Lorsque nous indiquons le nom du fichier que nous enregistrons (`save()`, `write.csv()`, `download.file()`, etc.), nous indiquons également où nous souhaitons l'enregistrer. Par défaut, R enregistre les fichiers dans notre "Working directory" = dossier de travail.
Quand nous travaillons dans un projet, le working directory correspond au dossier du projet. Lorsque ce n'est pas le cas, R a choisi à l'installation un working directory pour vous. Pour le trouver, utiliser la fonction `getwd()`.
Avoir un projet permet d'avoir plus de contrôle sur son working directory.

Téléchargement export (1)

Pour préciser l'emplacement du fichier, il faut ajouter le chemin avant par exemple:

```
download.file(url,
              destfile = "C:/Users/moi/Desktop/ofs.xlsx",
              mode = "wb")
```

Lorsque nous sommes dans un projet, il est assez facile de compléter ce champ en utilisant l'autocomplétion (appuyer sur TAB), nous pouvons ainsi enregistrer le fichier relativement au dossier du projet.

Exercice

Temps: ~5 minutes

Télécharger soit une image, soit un fichier en ligne avec la méthode présentée. Voici quelques endroits où trouver de quoi tester:

google image

OFS

Un peu de HTML

HTML: Langage de balisage qui permet de structurer une page web.

```
<!DOCTYPE html>
<html>
    <head>
        <title>C'est un fichier html<title>
    </head>
    <body>
        <table>
            (...)
        </table>
        <h1>C'est un grand titre</h1>
    </body>
<html>
```

les Tags

Les **tags** peuvent être considérés comme des balises qui permettent de savoir dans quelle partie nous nous trouvons et quel type d'information nous pouvons nous attendre à trouver. On les reconnait, car ils sont entourés de < et > et viennent souvent par pair identique.

Par exemple: `<head>` pour l'entête de la page web.

Il existe des tags très utilisés qui ont des fonctions spécifiques (voir page suivante).

Tags usuels

Tag	Utilisation
`<html>`	Base pour un fichier HTML
`<head>`	En-tête d'un fichier HTML
`<body>`	Contenu d'un fichier HTML
`<title>`	Titre du fichier HTML
`<h1>`, `<h2>`, ..., `<h6>`	Niveaux de titre du texte
`<p>`	Paragraphe de texte
`<a>`	Lien
`<img>`	Image
`<table>`	Tableau

Il y en a d'autres, `<div>`, `<span>`, `<li>`, etc.

CSS et sélecteur CSS

CSS: Langage qui sert à la présentation visuelle et l'édition de la page web (style, couleurs, etc.).

Sélecteur CSS: Code qui permet de sélectionner des parties d'un code HTML.

Lorsque nous avons analysé nos résultats avec la fonction `html_elements()`, nous avons spécifié les tags HTML "table" pour les tableaux et "h1", "h2", "h3" et "p" pour les titres et les paragraphes. Nous avons en réalité utilisé des sélecteurs CSS.

Code source d'une page web

Pour voir le code source d'une page web. Il suffit de faire une clique droit sur la page et trouver quelque chose comme "code source" dans les options.

Si nous voulons voir un élément particulier d'une page web, nous pouvons faire la même chose en cliquant sur l'élément, mais cette fois-ci sélectionner une option comme "inspecter". Nous obtenons donc le code HTML spécifique de l'objet.

Exercices

Choisir une page internet (par exemple un journal en ligne ou Wikipédia).

1. Inspecter son code source et essayer de trouver des balises comme h1, h2, h3, p et tables s'il y en a.
2. Inspecter la balise d'une partie spécifique du site web qui vous intrigue.
3. Importer la page HTML sur R et essayer de récupérer des éléments de votre choix.

Sélecteur CSS

Maîtriser les sélecteur CSS n'est pas si difficile. Mais ça peut demander du temps. Dans le cadre de ce cours, nous n'allons pas apprendre à les utiliser par manque de temps. Mais il existe une solution qui simplifie la vie et qui marche dans la quasi-totalité des cas: les sélecteurs automatiques.

Source pour apprendre les sélecteurs CSS:

- codeur.com

Si vous souhaitez continuer à plus haut niveau dans le web scraping, je vous recommande chaleureusement d'apprendre les sélecteurs CSS et le REGEX qui vous aideront dans n'importe quelle situation.

Sélecteurs automatiques

Pour Chrome, SelectorGadget est un bon choix:

- SelectoGadget

Pour Firefox, ScrapeMeta Beta fait du bon travail:

- ScrapeMeta Beta

Lorsque vous les avez installés, l'utilisation est très facile. Nous allons faire une démonstration avec des discussions sur le net.

Données non structurées

Note

Il n'est pas rare de trouver sur internet des données qui sont éparpillées sur une page. On souhaiterait les avoir ensemble. Nous devons donc les collecter et les joindre dans un objet commun (généralement un tableau de données).
C'est dans ce genre de moment que savoir utiliser les sélecteurs css ou les sélecteurs automatiques devient pratique.

Exemple avec un forum: Dans une discussion, nous voulons prendre pour chaque message l'auteur, la date et le contenu.

Site utilisé: Au jardin

```
url <- "https://www.aujardin.org/viewtopic.php?t=71654"
```

Collecter les informations (1)

Nom

```
nom <-
  page %>%
  html_nodes(".responsive-hide .username") %>%
  html_text2()
```

Date

```
time <-
  page %>%
  html_nodes("time") %>%
  html_text2()
```

Collecter les informations (2)

Message

```
message <-
  page %>%
  html_nodes("#page-body .content") %>%
  html_text2()
```

Créer un tableau de donnée

Base

```
forum <- data.frame(nom, time, message)
```

{dplyr}/{tibble}

```
library(dplyr)
forum <- tibble(nom, time, message)
```

Ensuite nous pouvons enregistrer les tableaux de données au format que l'on souhaite.

Export en un grand fichier texte

dplyr

```
texte2 <-
  forum %>%
  summarise(texte = paste(nom, time, message, sep = "\n")) %>%
  paste(collapse = "\n\n")

writeLines(texte2, con = "texte2.txt")
```

Nous verrons plus tard comment faire un export multiple.

Exercice

Temps: ~5 minutes

Créer un tableau de données avec les messages et leurs auteurs. Sauvegarder le résultat dans le format que l'on souhaite.

- Archive of our own (**Fanfiction**)
- bored of studies (**Étudiant.e.s**)
- forum tom's guide (**Informatique**)
- forum tom's hardware (**Informatique**)
- Au jardin (**Jardinage**)

Collecter des liens

Nous pouvons également collecter des liens. Cette partie est cruciale pour la suite du cours:

Liste de presse en ligne francophone:

Les annuaires

Parfois le sélecteur automatique ne fait pas du bon travail surtout lorsque nous avons des pages avec une très grande hiérarchie d'éléments.

Collecter des liens

La fonction `html_attr()` nous permet de préciser que nous cherchons un lien (“href”). La fonction `url_absolute()` nous permet de compléter tous les liens où il manque la base du site:

```
url <- "https://www.20minutes.fr/"

page <- read_html(url)

liens <-
  page %>%
  html_nodes("article a") %>%
  html_attr("href") %>%
  url_absolute("https://www.20minutes.fr/")
```

Automatisation

Automatisation

Note

Jusqu'à maintenant, nous nous étions intéressés à la collecte sur une seule page. Le web scraping a pour objectif de collecter des données sur de multiple page et de manière automatique pour nous éviter le travail manuel.

Requêtes

Requêtes

Pour pouvoir automatiser nos collectes, nous devons savoir ce qui se passe.

HTTP: HyperText Transfer Protocol

URL: Uniform Resource Locator

Requête HTTP

Demande de ressource vers un serveur du navigateur qui va formater les informations sur l'ordinateur. Les serveurs ont leur règle et leur limite.

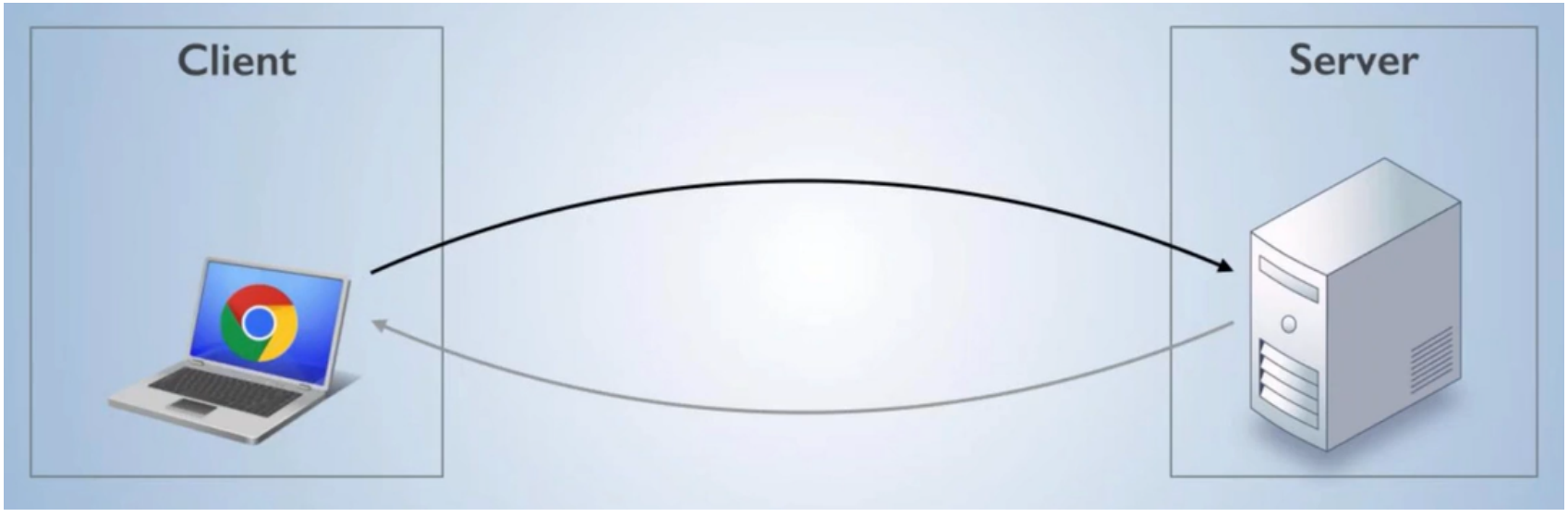


Figure 2: https://www.rstudio.com/resources/webinars/part-1-easy-ways-to-collect-different-types-of-data-from-the-web-with-r/

Limites des requêtes

Avec du code, nos ordinateurs sont capables d'effectuer plusieurs requêtes par seconde. Tous les serveurs ne sont pas forcément en mesure de gérer autant de requêtes. Il y a deux cas de figure qui peuvent arriver si on effectue trop de requêtes en un temps limité:

1. Le serveur nous bloque l'accès (cela peut durer plusieurs mois)
2. Nous faisons sauter le serveur et nous pouvons avoir de gros problèmes

Généralement, il est préférable d'espacer ses requêtes de 3 secondes pour ne pas avoir de problèmes s'il y a beaucoup de requêtes

Map (ou apply)

Map (ou apply)

Consiste à appliquer une fonction à chaque élément d'une liste. Il existe des fonctions de basent dans R (ex. `apply()`, `lapply()`, `sapply()`, etc.), mais le package {purrr} nous donne une plus grande variété de fonctions pour cette tâche.

Similaire à une boucle "for"

Nous faisons ici de la programmation fonctionnelle.

Fonctions

Il est nécessaire de savoir créer des fonctions. Pour cela, nous avons besoin au moins d'un nom (si non anonyme), d'un contenu. Par exemple la fonction suivante sert à dire bonjour:

```
bonjour <- function(){
  "bonjour"
}

bonjour()
## [1] "bonjour"
```

Fonctions

Nous pouvons ajouter un paramètre/variable qui peut servir à des opérations. Par exemple la fonction suivante permet d'ajout 2 à un chiffre:

```
plus2 <- function(x){
  x + 2
}

plus2(3)
## [1] 5
```

Fonctions

Il est préférable d'ajouter `return()` pour être sûr du résultat que la fonction retourne surtout s'il y a plusieurs éléments:

```
addition <- function(a, b){
  res <- a + b
  return(res)
}

addition(2, 2)
## [1] 4
```

Fonction + map (base)

Nous pouvons facilement appliquer ces fonctions sur une liste d'éléments.

```
chiffres <- list(1,2,3)
```

Base

```
lapply(chiffres, plus2)
## [[1]]
## [1] 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 5
```

Fonction + map ({purrr})

purrr

```
library(purrr)

map(chiffres, plus2)
## [[1]]
## [1] 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 5
```

Exercice

Temps: ~5 minutes

Créer vos propres fonctions (une avec 0, 1 et 2 paramètres) et testez-les d'abbord sur des élément uniques puis sur des listes ou vecteurs.

Pourquoi faire tout ça?

Pour pouvoir collecter des données sur plusieurs pages (automatisation), il faut:

1. Faire une ou plusieurs requêtes spécifiques
2. Collecter la liste des pages
3. Appliquer notre code de collecte à chaque page

Pour écrire le code, il est préférable d'aller dans l'autre sens.

Collecte d'articles d'un journal en ligne

Collecte d'articles d'un journal en ligne

Faire l'exercice ensemble

Choisir parmi la liste des journaux francophones:

les annuaires

Vérifier qu'il est possible de faire une recherche dans les articles sans compte et que le contenu des pages peut être collecté. Généralement, c'est bon dans la majorité des cas, mais ce n'est pas possible avec le journal 20 minutes de la Suisse. Si possible choisir des pages qui se tournent.

Fonction de collecte (1)

Ici c'est très simple, nous souhaitons simplement le texte. Nous testons un exemple:

```
url <-
  "https://www.lefigaro.fr/jardin/chat-male-ou-chat-femelle

page <- read_html(url)

page %>%
  html_nodes("h1, h2, h3, p") %>%
  html_text2() %>%
  paste(collapse = " ")
```

Fonction de collecte (2)

Nous transformons notre code en fonction:

```
collecter <- function(url){
  page <- read_html(url)

  texte <-
    page %>%
    html_nodes("h1, h2, h3, p") %>%
    html_text2() %>%
    paste(collapse = " ")

  return(texte)
}
```

Fonction de collecte (3)

Nous devons ajouter une pause de 3 secondes pour éviter la surcharge de requête avec la fonction `Sys.sleep()` (avant `return()`!):

```
collecter <- function(url){
  page <- read_html(url)

  texte <-
    page %>%
    html_nodes("h1, h2, h3, p") %>%
    html_text2() %>%
    paste(collapse = " ")

  Sys.sleep(3)
  return(texte)
}
```

Test (1)

Nous pouvons tester notre fonction sur une autre page:

```
collecter("https://www.lefigaro.fr/sciences/les-chats-s-int
## [1] "Rubriques et services du Figaro Le Figaro Rubrique
```

Test (2)

Nous pouvons tester sur plusieurs pages pour nous assurer que la pause marche:

```
liens_articles <- c("https://www.lefigaro.fr/sciences/les-c
  "https://www.lefigaro.fr/jardin/chat-male-ou-chat-femelle
  "https://www.lefigaro.fr/animaux/pourquoi-les-chats-n-air

articles <- map(liens_articles, collecter)
```

Liste de liens

Maintenant, il nous faut une liste de liens. C'est assez facile à faire en prenant le lien de la page de recherche:

```
url <- "https://recherche.lefigaro.fr/recherche/chat/"

page <- read_html(url)

liens_articles <-
  page %>%
  html_elements(".fig-profil-headline a") %>%
  html_attr("href")
```

Liste de liens en fonction

Nous pouvons transformer ce code en fonction pour le rendre généralisable et même ajouter une pause de 3 secondes pour plus de requêtes (on verra pourquoi plus tard):

```
lister <- function(url){
  page <- read_html(url)

  liens_articles <-
    page %>%
    html_elements(".fig-profil-headline a") %>%
    html_attr("href")

  Sys.sleep(3)

  return(liens_articles)
}
```

Test

Nous pouvons tester avec une autre page de requête:

```
liens_articles <- lister("https://recherche.lefigaro.fr/rec
```

Première automatisation

Nous pouvons maintenant mettre tout ça ensemble. Pour limiter la durée du travail, nous pouvons prendre les 3 premiers articles de "liens_articles" avec `[1:3]` (à retirer si vous voulez tous les articles):

```
url <- "https://recherche.lefigaro.fr/recherche/chat/"

liens_articles <- lister(url)

articles <- map(liens_articles[1:3], collecter)
```

Nous avons enfin réalisé notre première automatisation!

Spécification de la requête

Nous devons encore spécifier la requête dans notre travail. Nous pouvons le faire depuis l'URL:

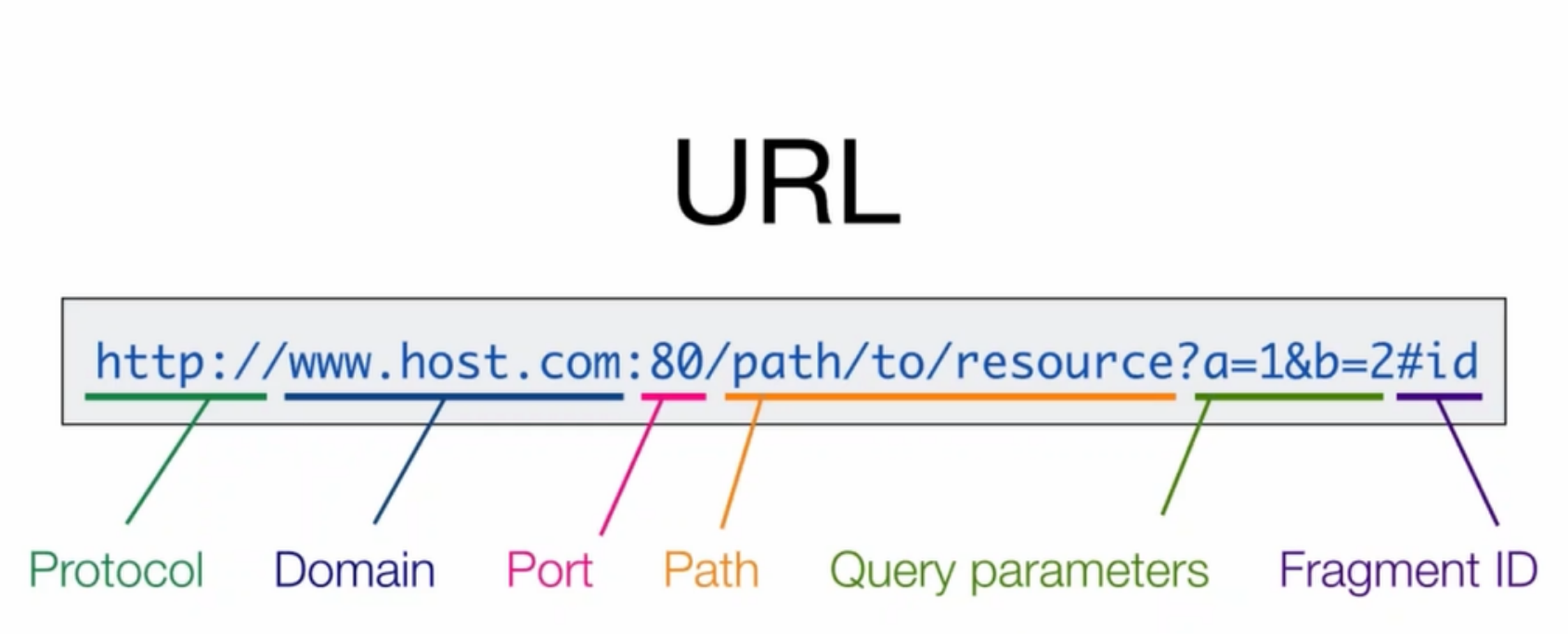


Figure 3: Structure d'une URL

Cas spécifique

Dans le cas du Figaro, nous pouvons facilement transformer la requête en remplaçant le mot "chat" par autre chose (ça peut être plus compliqué en fonction des sites):

```
"https://recherche.lefigaro.fr/recherche/chat/"
## [1] "https://recherche.lefigaro.fr/recherche/chat/"
"https://recherche.lefigaro.fr/recherche/chien/"
## [1] "https://recherche.lefigaro.fr/recherche/chien/"
```

Décomposition

Nous pouvons donc décomposer notre lien pour permettre de rechercher des éléments.

```
mot <- "économie"

base <- "https://recherche.lefigaro.fr/recherche/"

lien <- paste0(base, mot, "/")
```

Décomposition et fonction

Nous pouvons aussi transformer notre décomposition en fonction:

```
chercher <- function(mot){
  base <- "https://recherche.lefigaro.fr/recherche/"
  lien <- paste0(base, mot, "/")

  return(lien)
}
```

Test

```
chercher("livre")
## [1] "https://recherche.lefigaro.fr/recherche/livre/"
```

Total

Nous pouvons tout combiner:

```
url <- chercher("livre")

liens_articles <- lister(url)

articles <- map(liens_articles[1:3], collecter)
```

Nous pourrions transformer tout ça en une fonction. Mais il faut d'abord généraliser pour plusieurs pages de recherche.

Requête sur plusieurs pages (1)

Pour pouvoir faire des requêtes sur plusieurs pages, nous devons voir comment l'URL change lorsque nous tournons les pages. Malheureusement, ce n'est pas possible pour le journal Figaro. Ici nous faisons ça avec le site journal l'opinion. Nous voyons par exemple ce qui change en changeant de page et nous pouvons forcer ce style pour la première page:

Requête sur plusieurs pages (2)

```
# Page 1
"https://www.lopinion.fr/rechercher?q=chat&s=0"
## [1] "https://www.lopinion.fr/rechercher?q=chat&s=0"

# Page 2
"https://www.lopinion.fr/rechercher?q=chat&s=0&p=2"
## [1] "https://www.lopinion.fr/rechercher?q=chat&s=0&p=2"

# Page 3
"https://www.lopinion.fr/rechercher?q=chat&s=0&p=3"
## [1] "https://www.lopinion.fr/rechercher?q=chat&s=0&p=3"

# Page 1 forcée (marche)
"https://www.lopinion.fr/rechercher?q=chat&s=0&p=1"
## [1] "https://www.lopinion.fr/rechercher?q=chat&s=0&p=1"
```

Nouvelles fonctions (1)

L'avantage, c'est qu'il suffit de modifier le sélecteur css pour la fonction de collecte des liens...

```
lister <- function(url){
  page <- read_html(url)

  liens_articles <-
    page %>%
    html_elements(".Promo-title .Link") %>%
    html_attr("href")

  Sys.sleep(3)

  return(liens_articles)
}
```

Nouvelles fonctions (2)

Et la fonction de recherche:

```
# Exemple pour la page 1
chercher <- function(mot){
  base1 <- "https://www.lopinion.fr/rechercher?q="
  base2 <- "&s=0&p=1"

  lien <- paste0(base1, mot, base2)

  return(lien)
}

chercher("fleur")
## [1] "https://www.lopinion.fr/rechercher?q=fleur&s=0&p=1
```

Nouvelles fonctions (3)

Nous pouvons ajouter un paramètre/variable page pour indiquer le numéro de page. On ajouter "= 1" pour que la valeur de défaut soit 1 si on ne spécifie rien:

```
chercher <- function(mot, page = 1){
  base1 <- "https://www.lopinion.fr/rechercher?q="
  base2 <- "&s=0&p="

  lien <- paste0(base1, mot, base2, page)

  return(lien)
}
```

Nouvelles fonctions (3.5)

```
chercher("montre")
## [1] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
chercher("montre", 7)
## [1] "https://www.lopinion.fr/rechercher?q=montre&s=0&p="
```

Nouvelles fonctions (4)

Finalement, nous pouvons faire en sorte que la fonction nous crée plusieurs pages si on lui donne un numéro supérieur à 1 à l'aide d'un vecteur "1:" (pour 1 à...):

```
chercher <- function(mot, page = 1){
  base1 <- "https://www.lopinion.fr/rechercher?q="
  base2 <- "&s=0&p="

  lien <- paste0(base1, mot, base2, 1:page)

  return(lien)
}
```

Nouvelles fonctions (4.5)

```
chercher("montre")
## [1] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
chercher("montre", 7)
## [1] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
## [2] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
## [3] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
## [4] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
## [5] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
## [6] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
## [7] "https://www.lopinion.fr/rechercher?q=montre&s=0&p=
```

Total

On met de nouveau tout ensemble:

```
url <- chercher("livre")

liens_articles <- lister(url)

articles <- map(liens_articles, collecter)
```

L'avantage est que cette fois-ci, nous pouvons spécifier plus de pages à fouiller au moment de la requête:

```
url <- chercher("livre", 3)

liens_articles <- lister(url)

articles <- map(liens_articles, collecter)
```

Finalisation (1)

Nous pouvons finaliser en transformant tout en une seule fonction:

```
lopinion <- function(mot, page){
  url <- chercher(mot, page)

  liens_articles <- lister(url)

  articles <- map(liens_articles, collecter)

  return(articles)
}
```

Finalisation (2)

Nous avons maintenant une fonction qui nous permet de spécifier une requête pour obtenir le texte d'articles de journaux de l'opinion:

```
articles <- lopinion("musique", 3)
```

Nous avons vu les fondamentaux de la collecte de données en ligne (web scraping). Il reste toutefois un certain nombre de choses à connaître. Et surtout, il faudra pratiquer pour être sûr de bien mémoriser.

Pour aller plus loin

Il y a encore des choses qu'il faut maîtriser pour la programmation fonctionnelle. Par exemple les exceptions, le cas avec des données non structurées, le cas des tableaux de données, les tests unitaires, le refactoring, etc.

Vous trouverez plus d'informations dans les sources données dans le plan du cours et aussi sur notre chaîne YouTube.

Il y a encore d'audtres choses intéressantes à voir pour l'automatisation:

- Lancer son code en tâche de fond
- Planifier ses scripts sur R

-Planifier ses scripts avec GitHub

Les restrictions

Aspect légal

Aspect légal

Le web scraping n'est pas illégal en soi.

Tant que nous ne collectons pas des données personnelles ou soumises à la propriété intellectuelle, le web scraping reste légal. Dans le cadre d'analyse de données pour une recherche, ça ne pose pas de problème à condition d'être transparent sur la source de donnée et la méthode de collecte.

Les sites internet peuvent aussi avoir des consignes concernant le web scraping sur leur page qui se trouvent généralement dans leur robots.txt. Le package {polit} permet d'intégrer ces informations dans notre code ce qui nous évite de spécifier nous-mêmes les temps d'attente:

- {polite}

Restrictions volontaires

Restrictions volontaires

Les sites peuvent placer des restrictions pour empêcher des bots de collecter les données sur leur page:

- En empêchant de manipuler le contenu
- En demandant d'avoir un "header", généralement un navigateur
- En faisant des CAPTCHAS ou autres tests pour vérifier que c'est bien un humain qui fait la requête

JavaScript

JavaScript

Les sites modernes utilisent du contenu dynamique. Le problème est qu'il faut un navigateur pour pouvoir avoir accès au contenu dynamique. Donc avec {rvest} ce contenu est invisible.

Exercice

Temps: 5 minutes

Essayer de collecter les commentaires sous un article du 20 minutes suisse ou sous une vidéo YouTube. Vous pouvez aussi essayer de collecter le contenu d'un tweet sur Twitter.

Solution (pour le JavaScript)

Il faut passer par un navigateur qui peut être contrôlé par R. généralement, le package {RSelenium} est très utilisé. Mais il a beaucoup de dépendances (beaucoup d'installation).

Il y a également des solutions utilisant PhantomJS pour régler ce problème. Mais cela demande une certaine connaissance de JavaScript et il y a aussi une dépendance externe.

Il y a une solution récente qui est d'utiliser le package {chromote} qui permet de contrôler son navigateur web avec R. Il y a également un package en cours de développement appelé {hayallbaz} qui facilite l'utilisation de{chromote} pour le web scraping. Mais il faut avoir le navigateur chrome installé sur son ordinateur (ne gère pas d'autres navigateurs).

C'est la dernière solution que nous favoriserons.

Installer les packages

Installation

```
installed.packages("chromote")
remotes::install_github("rundel/hayalbaz")
```

Librairie

```
library(hayalbaz)
```

Démonstration

Avoir les auteurs des commentaires sous un article dans le journal 20 minutes:

```
session <- puppet$new("https://www.20min.ch/fr/story/rebec

element <- session$get_elements(".authorNickname")

element %>%
  html_text2()
## character(0)
```

Les RESTful APIs et autres sources de données

Qu'est-ce qu'une API?

Qu'est-ce qu'une API?

*Une **API** est une interface par laquelle un programme ou un site web communique avec un autre. Elles sont utilisées pour partager des données et des services. Et il y a des formats et types différents.*

*Une **API RESTful** est l'une des nombreuses façons dont les programmes, les serveurs et les sites web peuvent partager des données et des services. REST (Representational State Transfer) décrit les règles générales de représentation des données et des services par le biais de l'API afin que d'autres programmes puissent demander et recevoir correctement les données et les services qu'une API met à disposition.*

Source: Nasty Newt, Grepper

Les APIs

Les (RESTful) APIs permettent de faciliter la collecte de données en mettant en place des processus de collecte de donnée en amont. Il suffit alors d'exécuter de simple requête. C'est à peu près ce que nous avons fait lorsque nous avons créé notre fonction que nous pourrions d'ailleurs transformer en API si nous le souhaitions. Lorsqu'une requête est envoyée, l'API nous retourne un fichier au format JSON.

L'utilisation d'APIs avec R n'est pas compliquée, le package {httr} fait un très bon travail. Mais la manipulation de données JSON avec R n'est pas simple, mais si nous avons de bons packages pour ça ({jsonlite} et {tidyjson}).

Les réseaux sociaux

Il est possible de créer son propre script de scraping pour collecter les données des réseaux sociaux, mais c'est très coûteux en temps et en énergie et c'est parfois très lent.

La plupart des réseaux sociaux ont des APIs pour collecter les données (Twitter, Youtube, Reddit, Facebook, etc.). Mais elles demandent en général d'avoir un compte et une clé d'authentification pour laquelle il faut faire une demande qui doit être acceptée par le réseau. Elles ont l'avantage de faciliter la collecte, mais elles sont souvent limitées et demandent de l'argent pour passer cette barrière.

En général lorsqu'il y a une APIs, les sites web préfèrent qu'on l'utilise plutôt que de collecter nous-mêmes. Elles sont toujours accompagnées de documentation.

Les packages de collecte de R

Il y a des particuliers qui ont créé des packages autour de certaines APIs pour nous faciliter l'utilisation. Vous pouvez trouver une liste non exhaustive dans les sources du plan du cours sous "Packages R pour collecter des données".

Où trouver des données?

Où trouver des données?

Nous pouvons trouver beaucoup de données. Que ce soit dans les sites gouvernementaux, les repository GitHub, les réseaux sociaux, les articles de presse en ligne, etc. Il y a énormément de possibilités avec le web scraping.

Vous trouverez dans le plan du cours d'autres sources de données intéressantes.

Feedback

Lien votamatic

Code: DTBH

Merci pour votre attention!