Qu'est-ce que Pandas ?

Pandas est une bibliothèque open-source pour Python qui fournit des **structures de données haute performance** et des **outils d'analyse de données**.

Elle est particulièrement utile pour :

- **Manipulation de données** :
  - importation de données structurées : par exemple au format CSV, «*comma separated values*», générées par `tshark` ou bien du json retourné par un webservice ;
  - nettoyage : éliminer les informations incomplètes ou inutiles ;
  - transformation : changer de type, interprétation de dates ;
  - préparation des données pour l'analyse : sélectionner par condition ;
- **Analyse de données** :
  - statistiques descriptives : sur les valeurs numériques ou non ;
  - enrichissement : calculer de nouvelles informations et les ajouter ;
  - regroupement de données : regrouper les données par critères ;
- **Visualisation de données** :
  - intégration avec les bibliothèques comme **Matplotlib** ou **Seaborn** pour créer des visualisations.
  - création de rapport au format json, html ou csv.

Ce module doit être installé :

xterm
```
$ pip install requests
```

Il est nécessaire de créer et/ou gérer un environnement virtuel avec le module `venv`

- accès aux ressources par la méthode «classique» GET :

```
reponse = requests.get('https://p-fb.net')
```

- éventuellement avec des paramètres dans l'URL :

```
reponse = requests.get('https://monsite.org',
                               params={'lang':'fr','contenu':'json'})
```

- accès aux ressources par la méthode POST avec passage de paramètres :

```
reponse = requests('https://monsite.org', data = {'key' : 'value'})
```

- accès avec des headers choisis, comme pour passer un token d'autorisation :

```
reponse = requests('https://monsite.org',
                            headers = {'Authorization' : 'Bearer ##TOKEN##'})
```

Gestion de la réponse

L'objet `reponse` :

- `reponse.status_code` : le code de retour HTTP (200 Ok, 404 Not Found) ;
- `reponse.headers` : les entêtes de la réponse ;
- `response.content` : le contenu au format JSON, HTML ou binaire ;
  En utilisant, `reponse.json()` *on traite la réponse au format JSON.*

- au format CSV :

```
>>> df = pandas.read_csv('mes_donnees.csv')
```

- au format XML : Il faudra installer le module «`lxml`» :

```
xterm
$ pip install lxml
```

Ce qui permet de lire le contenu d'un fichier au format XML :

```
>>> df = pandas.read_xml('fichier.xml')
```

- convertir un dictionnaire :

```
>>> mon_dictionnaire = { 'col1' : [ 1, 2, 3 ], 'col2' : [ 4, 5, 6 ], 'col3' : [ 7, 8, 9 ] }
>>> df = pandas.DataFrame.from_dict(mon_dictionnaire)
>>> df
   col1  col2  col3
0     1     4     7
1     2     5     8
2     3     6     9
>>>
```

Les lignes sont numérotées chacune en commençant à 0 (colonne de gauche).

- convertir des données au format json :

```
>>> r = requests.get('http://ipinfo.io/')
>>> df = pandas.DataFrame(r.json(), index=[0])
>>> df
       ip                    hostname   city  ... postal      timezone              readme
0  10.10.10.10  10-10-10-10.dsl.ovh.fr  Paris  ...  75000  Europe/Paris  https://ipinfo.io/missin
gauth

[1 rows x 10 columns]
```

Il faut compléter les données avec un `index[0]`.

L'utilisation de `df = pandas.json_normalize(r.json())` *réalise le même travail.*

Sélectionner les colonnes


```
>>> df = pandas.DataFrame( {  'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ],
                              'Age' : [ 12, 18, 25, 25, 16 ] } )
>>> df
      Nom  Age
0  Pierre   12
1    Paul   18
2    Jean   25
3   Alice   25
4     Bob   16
```

Afficher les premières ligne de la table pandas :

```
>>> df.head(10) # les 10 premières
```

Afficher les dernières ligne de la table pandas :

```
>>> df.tail(10) # les 10 dernières
```

```
>>> df.Nom
0    Pierre
1      Paul
2      Jean
3     Alice
4       Bob
Name: Nom, dtype: object
```

```
>>> df['Nom']
0    Pierre
1      Paul
2      Jean
3     Alice
4       Bob
Name: Nom, dtype: object
```

On peut accèder à une colonne en utilisant directement le nom de la colonne, ou alors on peut la passer en chaine dans un accès à la manière d'un dictionnaire (obligatoire si le nom contient un espace).

```
>>> df[ [ 'Age', 'Nom'] ]
   Age     Nom
0   12  Pierre
1   18    Paul
2   25    Jean
3   25   Alice
4   16     Bob
```

On peut sélectionner plusieures colonnes en passant une liste.

Obtenir les données uniques d'une colonne :

```
>>> df.Age.unique()
array([12, 18, 25, 16])
```

Utilisation de la méthode `iloc`

```
>>> df.iloc[0]
Nom    Pierre
Age        12
Name: 0, dtype: object
```

▷ sélectionner un **élément** :

```
>>> df.iloc[0,0]
'Pierre'
```

▷ sélectionner une **tranche** :

```
>>> df.iloc[2:4]
     Nom  Age
2   Jean   25
3  Alice   25
```

On remarque que l'on conserve les valeurs d'index originales.

▷ sélectionner une «double tranche» ou un **sous tableau** :

```
>>> df.iloc[1:3, 1]
0    12
1    18
Name: Age, dtype: int64
```

▷ **modifier** une valeur :

```
>>> df.iloc[0,0] = 'Arthur'
>>> df
      Nom  Age
0  Arthur   12
1    Paul   18
2    Jean   25
3   Alice   25
4     Bob   16
```

Création d'index

Créer un **index** à partir d'une colonne :

```
>>> df
      Nom  Age
0  Arthur   12
1    Paul   18
2    Jean   25
3   Alice   25
4     Bob   16
>>> df.set_index('Nom', inplace = True)
        Age
Nom
Arthur   12
Paul     18
Jean     25
Alice    25
Bob      16
>>> p.loc['Jean']
Age    25
Name: Jean, dtype: int64
```

La méthode `loc` fonctionne maintenant sur cet index.

Réinitialiser l'index sur les données extraites :

```
>>> df
      Nom  Age
0  Arthur   12
1    Paul   18
2    Jean   25
3   Alice   25
4     Bob   16
>>> df = df.iloc[2:4]
>>> df
     Nom  Age
2   Jean   25
3  Alice   25
>>> df.reset_index(inplace = True, drop=True)
     Nom  Age
0   Jean   25
1  Alice   25
```

Accès aux éléments en lecture/écriture

Accès en lecture

Accès a une valeur en mode `[colonne][ligne]` :

```
>>> df = pandas.DataFrame( {  'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ],
                            'Age' : [ 12, 18, 25, 24, 16 ] } )
>>> df
      Nom  Age
0  Pierre   12
1    Paul   18
2    Jean   25
3   Alice   24
4     Bob   16
>>> df['Age'][2]
np.int64(25)
```

Accès avec la méthode `loc` en mode `[ligne, colonne]` :

```
>>> df.loc[2, 'Age']
np.int64(25)
```

Accès en écriture

On utilise l'opérateur `loc` :

```
>>> df.loc[2,'Age'] = 35
>>> df
      Nom   Age
0  Pierre  12
1    Paul  18
2    Jean  35
3   Alice  24
4     Bob  16
```

L'accès direct `df['Age'][2] = 35` est déconseillé car plus lent.

Le plus simple ? $\Longrightarrow$ Utiliser tout le temps la méthode `loc`.

Conversion du format de données

Obtenir la **liste des colonnes** et leur **type** :

```
df = pandas.DataFrame( { 'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ],
                         'Age' : [ 12, 18, 25, 24, 16 ] } )
>>> df.columns
Index(['Nom', 'Age'], dtype='object')
>>> df.dtypes
Nom       object
Age        int64
dtype: object
```

Conversion de **format numérique** :

```
>>> df['Age'] = df['Age'].astype(float)
>>> df
      Nom   Age
0  Pierre  12.0
1    Paul  18.0
2    Jean  35.0
3   Alice  24.0
4     Bob  16.0
>>> df.dtypes
Nom     object
Age    float64
dtype: object
```

Ici, on convertit les données d'entier vers flottant.

Conversion de **date** :

```
date = '22/Mar/2009:07:00:32 +0100'
temps = pandas.to_datetime(date, format='%d/%b/%Y:%H:%M:%S %z', utc = True)
>>> temps
Timestamp('2009-03-22 06:00:32+0000', tz='UTC')
```

Consulter les données incomplètes

Afficher les données où des informations sont manquantes :

```
>>> df = pandas.DataFrame( { 'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ], 'Age' : [ 12, 18,
25, 24, 16 ], 'Profession' : [ 'ingenieur', 'programmeur', 'programmeur', 'ingenieur', 'programmeur'
] } )
>>> nouvelle_df = pandas.DataFrame.from_dict({ 'Nom' : ['Albert'], 'Age' : [33] })
>>> pandas.concat([df,ndf])
      Nom  Age    Profession
0  Pierre   12     ingenieur
1    Paul   18  programmeur
2    Jean   25  programmeur
3   Alice   24     ingenieur
4     Bob   16  programmeur
0  Albert   33           NaN
>>> nouvelle_nouvelle_df = pandas.concat([df,ndf])
>>> nouvelle_nouvelle_df
      Nom  Age    Profession
0  Pierre   12     ingenieur
1    Paul   18  programmeur
2    Jean   25  programmeur
3   Alice   24     ingenieur
4     Bob   16  programmeur
0  Albert   33           NaN
>>>  nouvelle_nouvelle_df.isnull()
     Nom    Age  Profession
0  False  False       False
1  False  False       False
2  False  False       False
3  False  False       False
4  False  False       False
0  False  False        True
>>> nouvelle_nouvelle_df.isnull().sum()
Nom           0
Age           0
Profession    1
dtype: int64
```

numpy.nan ⟹ *valeur non numérique (Not A Number)*

Tableau de booléens

Si on veut sélectionner les éléments avec une valeur définie : `notnull()` au lieu de `isnull()`

- Ajouter une colonne vide :

```
>>> df = df.assign(nouvelle_colonne = None)
```

Le nom de la nouvelle colonne est exprimée sans être inclus dans une chaîne, et il faut mettre à jour la variable df pour que la table soit modifiée.

- Ajouter une ligne :

```
>>> nouvelle_ligne = pandas.DataFrame({ 'Colonne1' : [ 42 ], 'Colonne2' : [ 50 ] })
>>> pandas.concat([df, nouvelle_ligne], ignore_index = True)
```

Les colonnes non renseignées prendront la valeur `NaN` *(*`numpy.nan`*, testable avec* `pandas.isna()`*).*

- Supprimer une colonne :

```
>>> df.drop('nom_colonne', axis=1, inplace=True)
```

Le paramètre `inplace` *indique de modifier la table elle-même au lieu de retourner une copie.*

```
>>> df.drop(df.columns[1], axis=1, inplace=True)
```

La colonne est désignée par son index.

- Supprimer une ligne :

```
>>> df.drop(1, inplace = True)
```

Ici, c'est une ligne qui est supprimée par son indice.

- Supprimer une ligne par une condition :

```
>>> df = df[ df['Sauts'] > = 2]
```

Ici, on peut supprimer les lignes dont la valeur pour la colonne 'Saut' est inférieure à 2.

- Supprimer les lignes dont les données sont insuffisantes :

```
df.dropna(inplace = True)
```

Supprime les lignes pour lesquelles il y a une valeur `None` *dans une de ses colonnes.*

Création de nouvelles colonnes calculées


```
>>> df = pandas.DataFrame( { 'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ], 'Année embauche'
: [ 2018, 2019, 2016, 2020, 2017], 'Profession' : [ 'ingenieur', 'programmeur', 'programmeur', 'inge
nieur', 'programmeur' ] } )
>>> df
      Nom  Année embauche   Profession
0  Pierre            2018    ingenieur
1    Paul            2019  programmeur
2    Jean            2016  programmeur
3   Alice            2020    ingenieur
4     Bob            2017  programmeur
>>> df['Ancienneté'] = 2024 - df['Année embauche']
>>> df
      Nom  Année embauche   Profession  Ancienneté
0  Pierre            2018    ingenieur           6
1    Paul            2019  programmeur           5
2    Jean            2016  programmeur           8
3   Alice            2020    ingenieur           4
4     Bob            2017  programmeur           7
```

```
>>> mon_dictionnaire = { 'col1' : [ 1, 2, 3 ], 'col2' : [ 4, 5, 6 ], 'col3' : [ 7, 8, 9 ] }
>>> df = pandas.DataFrame.from_dict(mon_dictionnaire)
>>> df
   col1  col2  col3
0     1     4     7
1     2     5     8
2     3     6     9
>>> df['calcul'] = (df.col2 + df.col3)/ df.col1
>>> df
   col1  col2  col3  calcul
0     1     4     7    11.0
1     2     5     8     6.5
2     3     6     9     5.0
>>> df['calcul'] = 0
>>> df
   col1  col2  col3  calcul
0     1     4     7       0
1     2     5     8       0
2     3     6     9       0
```

écrasement des valeurs d'une colonne

Utilisation d'une fonction pour créer le contenu d'une nouvelle colonne

```
def ma_fonction(x):
  if x >= 18:
    return True
  return False
df = pandas.DataFrame( {  'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ],
                          'Age' : [ 12, 18, 25, 24, 16 ] } )
df['Majeur'] = df['Age'].apply(ma_fonction)
print(df)
```

Avec la méthode `apply`, on applique la fonction sur chaque valeur de la colonne indiquée et on construit une nouvelle colonne dont on a donné le nom lors de l'affectation.

xterm

```
   Nom  Age  Majeur
0  Pierre   12   False
1    Paul   18    True
2    Jean   25    True
3   Alice   24    True
4     Bob   16   False
```

On peut alors filtrer les données sur cette nouvelle colonne :

```
>>> df[df['Majeur'] == True]
     Nom  Age  Majeur
1   Paul   18    True
2   Jean   25    True
3  Alice   24    True
>>>
```

On peut ne pas utiliser de fonction dans le cas où la valeur est booléenne :

```
>>> selection = df['Age']>=18
>>> selection
0    False
1     True
2     True
3     True
4    False
Name: Age, dtype: bool
```

```
>>> df[selection]
     Nom   Age  Majeur
1   Paul  18.0    True
2   Jean  45.0    True
3  Alice  24.0    True
>>>
```

On obtient le même résultat.

Sélection des éléments d'une table par contenu ou expression régulière

Avec la méthode `.str.contains('...')` et une simple chaîne :

```
>>> selection = df['Nom'].str.contains('P')
>>> selection
0     True
1     True
2    False
3    False
4    False
Name: Nom, dtype: bool
>>> df[selection]
      Nom   Profession
0  Pierre    ingenieur
1    Paul  programmeur
```

Ici, seul 'Pierre' et 'Paul' contiennent un 'P'.

Utilisation d'**expression régulière** :

```
Series.str.contains(pattern, case=True, flags=0, na=nan, regex=True)
```

- ▷ `pattern` : séquence de caractères ou expressions régulières ;
- ▷ `case` : si True, recherche sensible à la casse. Si False, recherche insensible à la casse ;
- ▷ `flags` : Flags à passer directement au module re (par exemple : re.IGNORECASE).
- ▷ `na` : la valeur à mettre en cas de données manquantes ;
- ▷ `regex` : Si True, utilise `pattern` en expression régulière ou sinon comme une simple chaîne.

```
>>> selection = df['Nom'].str.contains(r'^.*i.*e$', regex=True)
>>> selection   # contient un True pour les noms contenant un 'i' et finissant par un 'e'
0     True
1    False
2    False
3     True
4    False
Name: Nom, dtype: bool
```

Sélection par conditions

```
df = pandas.DataFrame( { 'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ],
                         'Age' : [ 12, 18, 25, 24, 16 ],
          'Profession' : [ 'ingenieur', 'programmeur', 'programmeur', 'ingenieur', 'programmeur' ]} )
>>> df
      Nom  Age   Profession
0  Pierre   12    ingenieur
1    Paul   18  programmeur
2    Jean   25  programmeur
3   Alice   24    ingenieur
4     Bob   16  programmeur
```

Utilisation directe de la `dataframe` :

```
>>> df[(df.Age >= 18) & (df.Profession == 'programmeur')]
    Nom  Age   Profession
1  Paul   18  programmeur
2  Jean   25  programmeur
```

Utilisation de la méthode `query` :

```
>>> df.query('(Age>=18) & (Profession == "programmeur")')
    Nom  Age   Profession
1  Paul   18  programmeur
2  Jean   25  programmeur
```

Utilisation de la méthode `loc` :

```
>>> df.loc[(df.Age >= 18) & (df.Profession == 'programmeur')]
    Nom  Age   Profession
1  Paul   18  programmeur
2  Jean   25  programmeur
```

Utilisation de la méthode `iloc` :

```
>>> df.iloc[2:4]
     Nom  Age   Profession
2   Jean   25  programmeur
3  Alice   24    ingenieur
```

Incohérence de l'index

Lorsque vous sélectionnez des éléments d'une «*DataFrame*», vous obtenez une nouvelle «*DataFrame*»...
⟹ **Mais**, l'index est **inconsistent**, car il contient des «trous» pour chaque ligne de la table initiale non sélectionnée !

Il est nécessaire de réinitialiser l'index :

xterm
```
>>> df_selection = df_selection.reset_index()
```

Utilisation de données incomplètes

Si une ligne d'une «*DataFrame*» ne possède pas de valeur pour une colonne, il peut être impossible d'appliquer un traitement dessus...

Il est nécessaire de supprimer les lignes ne possédant pas de valeurs :

xterm
```
>>> df_sans_ligne_incomplete = df[ df['colonne'].notnull()]
```

Les valeurs absentes peuvent être indiquées comme «NaN»*,* «Not an Number»*, quand la colonne contient des valeurs numériques*.
⟹Il est aussi nécessaire de **réinitialiser l'index** avec «`reset_index()`»...

Statistiques numériques

On utilise la méthode `describe` qui ne traite que les données entières ou flottante :

```
>>> df
      Nom  Age   Profession
0  Pierre   12    ingenieur
1    Paul   18  programmeur
2    Jean   25  programmeur
3   Alice   24    ingenieur
4     Bob   16  programmeur
>>> df3.describe()
             Age
count   5.000000
mean   19.000000
std     5.477226
min    12.000000
25%    16.000000
50%    18.000000
75%    24.000000
max    25.000000
```

Pour traiter les données non numérique :

```
>>> df.describe(include='object')
           Nom   Profession
count        5            5
unique       5            2
top     Pierre  programmeur
freq         1            3
```

Compter le nombre d'occurences de chaque valeur unique avec `value_counts`

Sur une série :

```
import pandas

s = pandas.Series(['a', 'b', 'a', 'c', 'b', 'b'])
print(s.value_counts())  # Sortie: a 2, b 3, c 1
```

Pour une table :

```
>>> df = pandas.DataFrame( { 'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ], 'Profession' : [
'ingenieur', 'programmeur', 'programmeur', 'ingenieur', 'programmeur' ] } )
>>> df
      Nom   Profession
0  Pierre    ingenieur
1    Paul  programmeur
2    Jean  programmeur
3   Alice    ingenieur
4     Bob  programmeur
>>> df['Profession'].value_counts()
Profession
programmeur    3
ingenieur      2
Name: count, dtype: int64
>>>
```

Ou :

```
>>> df.value_counts(subset = 'Profession')
Profession
programmeur    3
ingenieur      2
Name: count, dtype: int64
```

Ici, on utilise `subset` pour sélectionner la colonne (on peut aussi donner une liste de colonnes `['colonne1', 'colonne2']` ce qui donne un comptage des différentes combinaisons uniques des valeurs de ces colonnes).

Regrouper les données et trier

Utilisation de la méthode `groupby` :

```
>>> df = pandas.DataFrame( { 'Nom' : [ 'Pierre', 'Paul', 'Jean', 'Alice', 'Bob' ], 'Age' : [ 12, 18,
25, 24, 16 ], 'Profession' : [ 'ingenieur', 'programmeur', 'programmeur', 'ingenieur', 'programmeur'
] } )
>>> df.groupby('Profession')['Age'].mean()
Profession
ingenieur      18.000000
programmeur    19.666667
Name: Age, dtype: float64
```

On calcule l'âge moyen des différentes professions.

Utilisation des méthodes `groupby` et `count` :

```
>>> df.groupby('Profession')['Profession'].count()
Profession
ingenieur      2
programmeur    3
Name: Profession, dtype: int64
```

Pour compter le nombre d'individus de chaque profession.

Pour trier suivant le contenu d'une colonne :

```
>>> df.sort_values('Age')
      Nom  Age   Profession
0  Pierre   12    ingenieur
4     Bob   16  programmeur
1    Paul   18  programmeur
3   Alice   24    ingenieur
2    Jean   25  programmeur
>>> ndf = df.sort_values('Age', ascending=False)
>>> ndf
      Nom  Age   Profession
2    Jean   25  programmeur
3   Alice   24    ingenieur
1    Paul   18  programmeur
4     Bob   16  programmeur
0  Pierre   12    ingenieur
```

On peut ensuite prendre les premières valeurs :

```
>>> ndf.iloc[0:2]
     Nom  Age   Profession
2   Jean   25  programmeur
3  Alice   24    ingenieur
```

Attention

Rénitialiser l'index du résultat :

```
ndf.reset_index()
```

Parcourir tous les éléments d'une table

Avec `iloc`

```
for ligne in range(len(df)):
  for colonne in range(len(df.columns)):
    print(df.iloc[ligne,colonne], " ", end = '')
  print()
```

xterm

```
Pierre  12  ingenieur
Paul  18  programmeur
Jean  25  programmeur
Alice  24  ingenieur
Bob  16  programmeur
```

Avec `itertuples`

```
for ligne in df.itertuples():
    print(ligne.Index, ":", ligne.Nom, ligne.Age)
```

xterm

```
0 : Pierre 12
1 : Paul 18
2 : Jean 25
3 : Alice 24
4 : Bob 16
```

Une variante :

```
for ligne in df.itertuples():
    print(ligne[0], ":", ligne[1], ligne[2])
```

xterm

```
0 : Pierre 12
1 : Paul 18
2 : Jean 25
3 : Alice 24
4 : Bob 16
```

Avec `iterrows`

```
for index,ligne in df.iterrows():
    print(index, ":", ligne["Nom"], ligne["Age"])
```

xterm

```
0 : Pierre 12
1 : Paul 18
2 : Jean 25
3 : Alice 24
4 : Bob 16
```

Exportation des tables

- au format **csv** :

```
>>> df.to_csv('mon_fichier.csv')
```

Crée un fichier.

- au format **json** :

```
>>> df.to_json()
```

Retourne un chaîne de caractères.

- au format **html** :

```
>>> df.to_html()
```

Retourne une chaîne de caractères.

Un serveur Web qui appelle la fonction `travail_pandas` sur `http://localhost:8000/` :

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class RequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(travail_pandas().encode('utf8'))

def run_server():
    server_address = ('', 8000)
    httpd = HTTPServer(server_address, RequestHandler)
    print("Server running at http://localhost:8000")
    httpd.serve_forever()

def travail_pandas():
    # Travail sur la table pandas
    return df.to_html()

run_server()
```

On récupére une liste de paquets depuis un fichier au format `pcap` :

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

from scapy.all import *
import pandas

# Extraction des donnees pertinentes des paquets
def traiter_paquet(packet):
    # on retourne un dictionnaire
    if packet.haslayer('IP'):
        return {
            'src_ip': packet['IP'].src,
            'dst_ip': packet['IP'].dst,
        }

les_paquets = rdpcap("paquets_captures.pcap")

# Extraction des infos sur tous les paquets IP
donnees = [ { 'IP' : p[IP].src } for p in les_paquets if IP in p]

# Creer une table
df = pandas.DataFrame(donnees)
print(df)
```

On veut obtenir la liste des adresses IP source de ces paquets et la tranformer en une table pandas :

Pour chaque paquet, il faut :

- ▷ extraire l'adresse IP source du paquet s'il contient un datagramme IP ;
- ▷ créer un dictionnaire pour associer un «label» associé à l'adresse IP source
- ▷ ajouter ce dictionnaire à une liste ;
  ⟹ On peut utiliser une «*list comprehension*» pour faire ce travail.
- ▷ donner cette liste à `pandas.DataFrame` pour créer la table : le label sera utilisé pour associer les éléments entre eux et nommer la colonne obtenue.

Lecture d'un fichier et extraction

Contenu du fichier à analyser dans pandas.

```
192.168.10.23 : Paul
192.168.17.87 : Pierre
192.168.47.98 : Jacques
192.168.34.11 : Alice
192.168.28.21 : Bob
```

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

import pandas
import sys
import re

nom_fichier = 'liste_machines.txt'
re_analyse = re.compile(r'^([.\d]+)\s*:\s*(.*)')

def traiter_ligne(l):
    resultat = re_analyse.search(l)
    if resultat:
        return { 'IP' : resultat.group(1), 'Utilisateur' : resultat.group(2) }
try:
    desc = open(nom_fichier, 'r');
except Exception as e:
    printf(e.args)
    sys.exit(1)

liste_infos = []
while 1:
    ligne = desc.readline()
    if not ligne:
        break
    ajout = traiter_ligne(ligne)
    if ajout:
        liste_infos.append(traiter_ligne(ligne))
df = pandas.DataFrame(liste_infos)
print(df)
```

On désire analyser un fichier `pcap` contenant plus de 90 000 paquets et déterminer les adresses IP source uniques :

```
>>> lp = rdpcap('nitroba.pcap')
>>> liste_adresses_ip_source = [p[IP].src for p in lp if IP in p]
>>> import pandas
>>> df = pandas.DataFrame({'IP' : liste_adresses_ip_source})
>>> df
                IP
0      192.168.1.64
1      74.125.19.83
2      192.168.1.64
3      74.125.19.19
4      74.125.19.19
...             ...
90374  192.168.15.1
90375  192.168.15.1
90376  192.168.15.1
90377  192.168.15.1
90378  192.168.15.4

[90379 rows x 1 columns]
>>> len(df.IP.unique())
432
```

Il faut :

- ▷ récupérer la liste de ces adresses IP source ;
- ▷ créer un dictionnaire avec cette liste (le nom de la colonne sera `IP` ;
- ▷ créer la table pandas en passant ce dictionnaire à `pandas.DataFrame` et compter les adresses uniques.

La méthode `to_frame` permet de créer une colonne et de l'ajouter à la table :

```
>>> k = df.groupby('Ik=df.groupby('IP')['IP'].count().to_frame('c')
>>> k
                  c
IP
10.0.1.5          8
116.252.234.84    1
12.129.147.65    45
12.129.210.41     4
12.129.210.46    10
...              ..
90.26.122.118     1
91.121.109.197   48
91.65.135.197     1
93.103.36.196     1
98.220.46.34      1

[432 rows x 1 columns]
>>> k.sort_values('c',ascending=False)
                      c
IP
192.168.15.4      34554
192.168.1.64       6818
208.111.148.6      3731
69.22.167.215      3635
74.125.15.159      3033
...                 ...
125.198.166.16        1
125.211.216.53        1
125.211.198.10        1
125.215.223.210       1
79.131.4.159          1

[432 rows x 1 columns]
```

la méthode `to_frame`
avec le nom de la nouvelle colonne

On peut ainsi appliquer un tri pour connaitre
quelles sont les adresses les plus présentes.

$\Longrightarrow$ La colonne `count` est ajoutée à la table.

Pour regrouper des colonnes dans une même table

On a :
- ▷ récupérer les adresses sources des différents datagramme IP et construit la table `df` ;
- ▷ récupérer les adresses destinations et construit la table `df2` ;
- ▷ on peut regrouper les deux colonnes dans une seule table :

```
>>> liste_adresses = [{ 'IP src' : p[IP].src} for p in lp if IP in p]
>>> liste_adresses_2 = [{ 'IP dst' : p[IP].dst} for p in lp if IP in p]
>>> df = pandas.DataFrame(liste_adresses)
>>> df2 = pandas.DataFrame(liste_adresses_2)
>>> df['IP dst'] = df2['IP dst']
>>> df
              IP src           IP dst
0       192.168.1.64     74.125.19.83
1       74.125.19.83     192.168.1.64
2       192.168.1.64     74.125.19.19
3       74.125.19.19     192.168.1.64
4       74.125.19.19     192.168.1.64
...              ...              ...
90374   192.168.15.1  239.255.255.250
90375   192.168.15.1  239.255.255.250
90376   192.168.15.1  239.255.255.250
90377   192.168.15.1  239.255.255.250
90378   192.168.15.4     74.125.19.99
```

***Attention** : il faut être sûr que les index des deux tables font correspondre les bonnes données entre elles !*

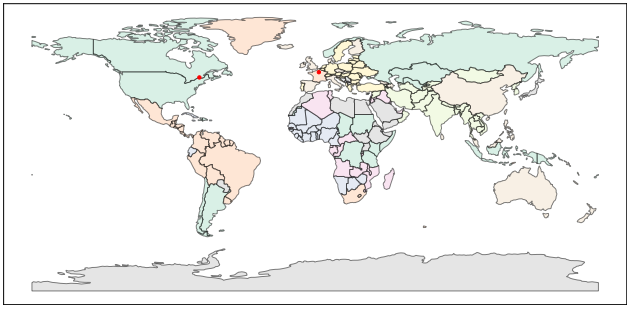
On peut aussi utiliser un `join` avec : `>>> df.join(df2)`

Il est plus sûr de faire l'extraction directement depuis les paquets :

```
>>> liste_adresses = [{ 'IP src' : p[IP].src, 'IP dst' : p[IP].dst } for p in lp if IP in p]
>>> df = pandas.DataFrame(liste_adresses)
>>> df
              IP src           IP dst
0       192.168.1.64     74.125.19.83
1       74.125.19.83     192.168.1.64
2       192.168.1.64     74.125.19.19
3       74.125.19.19     192.168.1.64
4       74.125.19.19     192.168.1.64
...              ...              ...
90374   192.168.15.1  239.255.255.250
90375   192.168.15.1  239.255.255.250
90376   192.168.15.1  239.255.255.250
90377   192.168.15.1  239.255.255.250
90378   192.168.15.4     74.125.19.99
[90379 rows x 2 columns]
```

Pour afficher des coordonnées (Lattitude,Longitude)


```
import geopandas
import pandas
import matplotlib.pyplot

world_file_path = 'ne_110m_admin_0_countries.shp'
gdf_world = geopandas.read_file(world_file_path)

df = pandas.DataFrame(
    {
        "Latitude": [48.8534, 45.5088],
        "Longitude": [2.3488, -73.5878],
    }
)

gdf_coordonnees = geopandas.GeoDataFrame(
    df, geometry=geopandas.points_from_xy(df.Longitude, df.Latitude)
)

fig, ax = matplotlib.pyplot.subplots(figsize=(20,10))
ax.set_xticks([])
ax.set_yticks([])
gdf_world.plot(ax = ax, aspect = 'auto', color = "lightgray", cmap = "Pastel2", edgecolor = "black",
alpha = 0.5)
gdf_coordonnees.plot(ax = ax, color = "red")

matplotlib.pyplot.title("Basic Map of World with GeoPandas")
matplotlib.pyplot.show()
```



Pour télécharger la carte du monde :

`https://www.naturalearthdata.com/http//www.naturalearthdata.com/download/110m/cultural/ne_110m_admin_0_countries.zip`

On peut créer un fichier au format csv avec tshark :

xterm

```
tshark -r mon_fichier.pcap -T fields -e frame.number -e frame.time_epoch -e eth.src -e eth.dst -e
eth.type -e ip.src -e ip.dst -e ip.proto -E header=y -E separator=, -E quote=d > sortie.csv
```

xterm

```
>>> import pandas
>>> df = pandas.read_csv('output.csv')
>>> df
       frame.number  frame.time_epoch            eth.src            eth.dst eth.type
ip.src           ip.dst  ip.proto
0                 1      1.216691e+09  00:1d:d9:2e:4f:61  00:1d:6b:99:98:68   0x0800
192.168.1.64     74.125.19.83       6.0
:::             .::.       ...    ...                  ...                 ...      ...
94409         94410      1.216707e+09  00:17:f2:e2:c0:ce  00:1d:d9:2e:4f:60   0x0800
192.168.15.4     74.125.19.99       6.0

[94410 rows x 8 columns]
# Conversion des dates au format epoch vers un format "humain"
>>> df['frame.time_epoch'] = pandas.to_datetime(df['frame.time_epoch'], unit = 's')
>>> df.iloc[0]
frame.number                                1
frame.time_epoch    2008-07-22 01:51:07.095278025
eth.src                          00:1d:d9:2e:4f:61
eth.dst                          00:1d:6b:99:98:68
eth.type                                   0x0800
ip.src                               192.168.1.64
ip.dst                               74.125.19.83
ip.proto                                      6.0
Name: 0, dtype: object
>>> df['eth.type'].unique()
array(['0x0800', '0x0806', nan], dtype=object)
>>> df[df['eth.type'].isna()]
       frame.number              frame.time_epoch            eth.src            eth.dst eth.type
ip.src ip.dst  ip.proto
18792         18793 2008-07-22 04:29:43.273530006  00:14:d1:44:a0:f1  ff:ff:ff:ff:ff:ff      NaN
NaN    NaN       NaN
```

Utilisation de `tshark`

On peut utiliser `tshark` pour récupérer le contenu des requêtes http :

xterm

```
>>> import pandas
>>> df = pandas.read_csv('output.csv')
>>> df2 = df[df['http.request.method'].notna()]
>>> df2[df2['http.request.method'].str.contains('GET')]
       frame.number  frame.time_epoch  ... http.request.method
http.request.uri
2                 3      1.216691e+09  ...                 GET
/mail/?logout&hl=en
171             172      1.216691e+09  ...                 GET
/a/hBIhP7YAbeh5-B7SEoEBNJqOT.AcGxgqbm/spacer.gif
:::             ...               ...  ...                 ...
94227         94228      1.216707e+09  ...                 GET
/firefox?client=firefox-a&rls=org.mozilla:en-U...
94239         94240      1.216707e+09  ...                 GET
/firefox?client=firefox-a&rls=org.mozilla:en-U...
[4462 rows x 10 columns]
>>> df2[df2['http.request.method'].str.contains('POST')]
       frame.number  frame.time_epoch  ... http.request.method
http.request.uri
10216         10217      1.216692e+09  ...                POST
/m57jean
10240         10241      1.216692e+09  ...                POST
/m57jean
:::             ...               ...  ...                 ...
90503         90504      1.216707e+09  ...                POST
/notifyft
94262         94263      1.216707e+09  ...                POST
/
[372 rows x 10 columns]
```

On notera la méthode `notna` permettant de sélectionner les lignes contenant une requête.