L1-S2 : UE Simulations numériques

SEANCE 10

Statistiques et module Pandas

19 avril 2022

Fonctions statistiques de numpy

Numpy intègre un ensemble de fonctions statistiques pouvant être appliquées à un tableau (array) :

- `np.amin(a)` : renvoie la valeur minimale du tableau `a`.
- `np.amax(a)` : renvoie la valeur maximale du tableau `a`.
- `np.percentile(a,q,axis=None)` : renvoie le q-ième percentile du tableau `a`.

Exemple : `np.percentile(a,30)` renvoie la valeur x pour laquelle 30% des valeurs de a sont inférieures à x.

Fonctions statistiques de numpy

- `np.mean(a)` : renvoie la moyenne des valeurs de a :

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

- `np.median(a)` : renvoie la médiane des valeurs de a, i.e. la valeur x pour laquelle la moitié des valeurs de a est inférieure à x et l'autre moitié supérieure à x. Cette fonction est donc équivalente à `np.percentile(a,50,axis=None)`
- `np.std(a)` : renvoie l'écart type des valeurs de a :

$$\sigma x = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N}}$$

Fonctions statistiques de numpy

In [1]:

```
1 import numpy as np
2
3 notes = np.array([19, 10, 14, 15, 8, 18])
4 print(np.mean(notes))
5 print(np.median(notes))
6 print(np.std(notes))
```

Out[1]:

```
14.0
14.5
3.958114029012639
```

Fonctions statistiques de numpy

Numpy permet des analyses statistiques raffinées sur des tableaux de données numériques du même type. Cependant, il est courant de travailler avec des tableaux qui contiennent des données de types différents et dans lesquels des cases peuvent être vides :

Animal	Description	Price ($)
Gnat	per gram	13.65
	each	0.01
Gnu	stuffed	92.50
Emu	stuffed	33.33
Armadillo	frozen	8.99

Ces structures ne peuvent pas être analysées simplement en utilisant le module numpy.

Dictionnaires

Les dictionnaires sont un type de base de Python, ils permettent d'associer un élément à une clef.
On appelle les éléments par leur clef et non pas par leur indice.

- Les dictionnaires sont délimités par des accolades.
- Clef et élément sont reliés par le symbole deux points (:).
- Un dictionnaire admet différents types pour les clefs et les éléments.

In [2]:
```
1 s = {"a": 3, 0:43, "c":[3,4,23], "d":"bla"}
2 print(s)
```

Out[2]:
```
{'a': 3, 0: 43, 'c': [3, 4, 23], 'd': 'bla'}
```

In [3]:
```
1 print(type(s))
```

Out[3]:
```
<class 'dict'>
```

Dictionnaires

In [4]:
```
1 print(len(s))
```

Out[4]:
```
4
```

In [5]:
```
1 print(s["c"])
```

Out[5]:
```
[3, 4, 23]
```

In [6]:
```
1 print(s["a"])
```

Out[6]:
```
3
```

Remarque : une clef numérique n'indique pas une position !

In [7]:
```
1 print(s[0])
```

Out[7]:
```
43
```

Dictionnaires

On peut modifier, ajouter ou retirer des termes d'un dictionnaire :

In [8]:
```
d={"a":1,"b":2}
d["b"]=3
print(d)
```

Out[8]:
```
{'a': 1, 'b': 3}
```

In [9]:
```
d["c"]=10
print(d)
```

Out[9]:
```
{'a': 1, 'b': 3, 'c': 10}
```

In [10]:
```
del d["a"]
print(d)
```

Out[10]:
```
{'b': 3, 'c': 10}
```

Dictionnaires

In [11]:
```
1 d.pop('c')
```

Out[11]:
```
10
```

In [12]:
```
1 print(d)
```

Out[12]:
```
{'b': 3}
```

Dictionnaires

On accède aux clefs et aux termes avec les méthodes `.keys()` et `.items()` :

In [13]:
```
1 s={"a":3, 0:43, "c":[3,4,23], "d":"bla"}
2 print(s.keys())
```

Out[13]:
```
dict_keys(['a', 0, 'c', 'd'])
```

In [14]:
```
1 print(s.items())
```

Out[14]:
```
dict_items([('a', 3), (0, 43), ('c', [3, 4,
    23]), ('d', 'bla')])
```

Dictionnaires

On peut parcourir un dictionnaire :

In [15]:
```
for c in s :
  print("clef",c,"terme",s[c])
```

Out[15]:
```
clef a terme 3
clef 0 terme 43
clef c terme [3, 4, 23]
clef d terme bla
```

La librairie Pandas

La librairie Pandas, basée sur Numpy, est un outil très performant pour manipuler et analyser des données. Elle permet de traiter des tableaux non homogènes avec éventuellement des cases vides. Elle intègre des outils de manipulation de données, d'analyse statistique et de visualisation.

In [16]:
```
1 import pandas as pd
```

Series

Une série est un tableau unidimensionnel pouvant contenir n'importe quel type de donnée. Chaque élément est repéré par un indice associé à sa position dans le tableau :

In [17]:
```
s = pd.Series([3,-5,7,4])
print(s[0])
```

Out[17]:
```
3
```

On peut aussi attribuer des indices de type string :

a	3
b	-5
c	7
d	4

In [18]:
```
s=pd.Series([3,-5,7,4], index=['a','b','c','d'
    ])
```

Series

Contrairement aux dictionnaires, on appelle les éléments d'une série par leur position ou par leur indice :

In [19]:
```
print("s[0] =", s[0], ", s['c'] =", s['c'], ",
      s[2] =", s[2])
```

Out[19]:
```
s[0] = 3 , s['c'] = 7 , s[2] = 7
```

On peut utiliser les techniques de slicing en utilisant les positions :

In [20]:
```
print(s[0::2])
```

Out[20]:
```
a    3
c    7
```

Series

Mais il est également possible de faire du slicing en utilisant les indices contenus dans l'index :

In [21]:
```
1 print(s['a':'c'])
```

Out[21]:
```
a     3
b    -5
c     7
```

ainsi que faire des opérations sur les données de la série :

In [22]:
```
1 (s['a':'c']+1)**2
```

Out[22]:
```
a    16
b    16
c    64
```

DataFrame

Un DataFrame correspond à un tableau 2D avec des étiquettes attribuées à chaque colonne

Country	**Capital**	**Population**
Belgium	Brussels	11190846
India	New Delhi	1303171035
Brazil	Brasilia	207847528

DataFrame

On peut facilement construire un dataframe à partir d'un dictionnaire :

```
data = {'Country': ['Belgium', 'India', 'Brazil'],
        'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
        'Population': [11190846, 1303171035, 207847528]}

df = pd.DataFrame(data)
```

à partir de séries :

```
country = pd.Series(["Belgium","India","Brazil"])
capital = pd.Series(["Brussels","New Delhi","Brasilia"])
population = pd.Series([11190846, 1303171035, 207847528])
df=pd.DataFrame({"Country":country, "Capital":capital,
                 "Population":population })
```

ou à partir d'un tableau numpy en donnant des noms aux lignes et aux colonnes :

```
df_numpy = pd.DataFrame(np.random.rand(3, 2),
                        columns=['foo', 'bar'],
                        index=['a', 'b', 'c'])
```

DataFrame

Sans spécification, un indice numérique est attribué à chaque ligne :

In [23]:
```
print(df)
```

Out[23]:
```
      Country     Capital  Population
0     Belgium    Brussels    11190846
1       India  New Delhi  1303171035
2      Brazil   Brasilia   207847528
```

On peut appeler les colonnes d'un dataframe par leur nom, on obtient ainsi une série :

In [24]:
```
print(df['Country'])
```

Out[24]:
```
0      Belgium
1        India
2       Brazil
```

ou simplement :

In [25]:
```
print(df.Country)
```

DataFrame

On peut extraire une série en appelant son indice dans le dataframe avec la méthode `.iloc[indice]` :

In [26]:
```
1 print(df.iloc[0])
```

Out[26]:
```
Country          Belgium
Capital         Brussels
Population      11190846
```

Cependant la méthode `.iloc` n'est valable que si l'index est constitué d'entiers. Si ce n'est pas le cas, il faut utiliser la méthode `.loc[]` par exemple si on a utilisé le nom des pays en index :

In [27]:
```
1 df_new = df.set_index("Country")
2 print(df_new.loc["Belgium"])
```

Out[27]:
```
Capital         Brussels
Population      11190846
Name: Belgium , dtype: object
```

DataFrame

Comme pour les Series on peut utiliser les techniques de slicing sur un DataFrame :

In [28]:
```
1 print(df[0:2])
```

Out[28]:
```
      Country     Capital  Population
0     Belgium    Brussels    11190846
1       India   New Delhi  1303171035
```

Il est possible de coupler le slicing avec la méthode de clef-indice :

In [29]:
```
1 print(df[0:2]["Country"])
```

Out[29]:
```
0    Belgium
1    India
```

In [30]:
```
1 print(df["Country"][2])
```

Out[30]:
```
Name: Country, dtype: object
Brazil
```

Importer des données

Pandas permet d'importer/exporter directement des bases de données dans différents formats de fichiers : csv, xls, json, sql...

- pd.read_excel()
- pd.read_csv()

Par défaut, la première ligne du fichier donne les noms des colonnes. Exemple pour un fichier csv :

In [31]:
```
1 !cat data/countries.csv
```

Out[31]:
```
# Fichier data/countries.csv
Country,Capital,Population
Belgium,Brussels,11190846
India,New Delhi,1303171035
Brazil,Brasilia,207847528
```

Importer des données

In [32]:
```
1 df=pd.read_csv("data/countries.csv")
2 print(df)
```

Out[32]:
```
   Country     Capital  Population
0  Belgium    Brussels    11190846
1    India  New Delhi  1303171035
2   Brazil    Brasilia   207847528
```

On peut connaître le nom des colonnes par l'attribut `.columns` :

In [33]:
```
1 df=pd.read_csv("data/countries.csv")
2 print(df.columns)
```

Out[33]:
```
Index(['Country', 'Capital', 'Population'],
      dtype='object')
```

Importer des données

Pandas gère aussi automatiquement les données manquantes :

In [34]:
```
!cat data/countries_missing.csv
```

Out[34]:
```
# Fichier data/countries_missing.csv
Country,Capital,Population
Belgium,,11190846
India,New Delhi,1303171035
Brazil,Brasilia
```

In [35]:
```
df=pd.read_csv("data/countries_missing.csv")
print(df)
```

Out[35]:
```
   Country     Capital    Population
0  Belgium         NaN  1.119085e+07
1    India  New Delhi  1.303171e+09
2   Brazil   Brasilia           NaN
```

Importer des données

On peut utiliser une colonne spécifique dans le fichier pour donner des noms aux lignes :

In [36]:
```
df=pd.read_csv("data/countries.csv", index_col='Country')
print(df.columns)
```

Out[36]:
```
Index(['Capital','Population'], dtype='object')
```

In [37]:
```
print(df.index)
```

Out[37]:
```
Index(["Belgium","India","Brazil"], dtype='object')
```

In [38]:
```
print(df.loc['Belgium'])
```

Out[38]:
```
Capital        Brussels
Population     11190846
Name: Belgium, dtype: object
```

Importer des données

Pandas permet de fusionner différents dataframes en utilisant les noms des colonnes. Le paramètre axis=1 permet d'ajouter des colonnes :

In [39]:
```
1 !cat data/countries.csv
```

Out[39]:
```
# Fichier data/countries.csv
Country,Capital,Population
Belgium,Brussels,11190846
India,New Delhi,1303171035
Brazil,Brasilia,207847528
```

In [40]:
```
1 !cat data/countries2.csv
```

Out[40]:
```
# Fichier data/countries2.csv
Country, Continent
Brazil,America
Belgium, Europe
India,Asia
```

Importer des données

In [41]:
```
1 c1=pd.read_csv("data/countries.csv",index_col="
      Country")
2 c2=pd.read_csv("data/countries2.csv",index_col=
      "Country")
3 d=pd.concat([c1,c2],axis=1)
4 print(d)
```

Out[41]:

	Capital	Population	Continent
Belgium	Brussels	11190846	Europe
Brazil	Brasilia	207847528	America
India	New Delhi	1303171035	Asia

Importer des données

Le paramètre axis=0 permet d'ajouter des lignes :

In [42]:
```
!cat data/countries3.csv
```

Out[42]:
```
# Fichier data/countries3.csv
Country,Capital,Population
France,Paris,67400000
Germany,Berlin,83186719
```

In [43]:
```
c1=pd.read_csv("data/countries.csv",index_col="
    Country")
c2=pd.read_csv("data/countries2.csv",index_col=
    "Country")
d=pd.concat([c1,c2],axis=0)
print(d)
```

Out[43]:

	Capital	Population
Belgium	Brussels	11190846
Brazil	Brasilia	207847528
India	New Delhi	1303171035
France	Paris	67400000
Germany	Berlin	83186719

Analyser les données

pandas permet une analyse rapide de la structure des données :

In [44]:
```
1 df.info()
```

Out[44]:
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
Country        3 non-null object
Capital        3 non-null object
Population     3 non-null int64
dtypes: int64(1), object(2)
memory usage: 152.0+ bytes
```

In [45]:
```
1 df.shape
```

Out[45]:
```
(3, 3)
```

In [46]:
```
1 df.columns
```

Out[46]:
```
Index(['Country', 'Capital', 'Population'],
      dtype='object')
```

Analyser les données

Ainsi que des outils statistiques intégrés :

In [47]:
```
df["Population"].mean()
```

Out[47]:
```
507403136.3333333
```

In [48]:
```
df["Population"].median()
```

Out[48]:
```
207847528.0
```

In [49]:
```
df["Population"].std()
```

Out[49]:
```
696134594.7840002
```

In [50]:
```
df["Population"].sum()
```

Out[50]:
```
1522209409
```

Analyser les données

In [51]:
```
df["Population"].min()
```

Out[51]:
```
11190846
```

In [52]:
```
df["Population"].max()
```

Out[52]:
```
1303171035
```

In [53]:
```
df.describe()
```

Out[53]:
```
       Population
count   3.000000e+00
mean    5.074031e+08
std     6.961346e+08
min     1.119085e+07
25%     1.095192e+08
50%     2.078475e+08
75%     7.555093e+08
max     1.303171e+09
```

Analyser les données

On peut également chercher l'indice d'une valeur particulière, par exemple ici la valeur maximale. La méthode .idxmax() renvoie l'index correspondant à la première occurrence.

In [54]:
```
1 df['Population'].max()
```

Out[54]:
```
1303171035
```

In [55]:
```
1 df['Population'].idxmax()
```

Out[55]:
```
'India'
```

In [56]:
```
1 df.loc[df['Population'].idxmax()] ['Capital']
```

Out[56]:
```
'New Delhi'
```

Extraire des données

Il est possible de faire des tests sur une colonne d'un DataFrame :

In [57]:
```
df["Population"]< 1e9
```

Out[57]:
```
Country
Belgium      True
India       False
Brazil       True
Name: Population, dtype: bool
```

Ce test permet ensuite d'extraire les lignes pour lesquelles le test est vrai :

In [58]:
```
df[df["Population"]< 1e9]
```

Out[58]:

	Country	Capital	Population
0	Belgium	Brussels	11190846
2	Brazil	Brasilia	207847528

Remarque : il est possible d'utiliser la méthode `.loc[]` avec une condition sur une colonne (ex : `df.loc[df.Population<1e9]`)

Extraire des données

Il est possible d'utiliser plusieurs conditions à la fois :

In [59]:
```
df[(df.Population<1e9) & (df.Population>1e8)]
```

Out[59]:
```
    Country    Capital  Population
2    Brazil   Brasilia   207847528
```

Attention, il faut utiliser l'opérateur binaire & et non l'opérateur logique and :

In [60]:
```
df[(df.Population<1e9) and (df.Population>1e8)]
```

Out[60]:
```
ValueError: The truth value of a Series is
    ambiguous. Use a.any() or a.all().
```

Remarque : cela est également le cas lorsque l'on utilise plusieurs conditions dans un numpy array.

Extraire des données

Opérateurs binaire :

Booléen	Binaire
and	&
or	|
not	~

Attention, les parenthèses sont obligatoires ! Elles permettent de contrecarrer l'ordre de préséance des opérateurs binaires sur les opérateurs conditionnels $<$ et $>$.

In [61]:
```
1 df[df.Population<1e9 & df.Population>1e8]
```

Out[61]:
```
TypeError: Cannot perform 'rand_' with a dtyped
    [int64] array and scalar of type [bool]
```

Extraire des données

La commande groupby() permet de former des groupes qui ont la même valeur dans une colonne

In [62]:
```
df_fruits = pd.read_csv("data/fruits.csv")
print(df_fruits)
```

Out[62]:

	color	fruit	price
0	red	cherry	3
1	yellow	banana	5
2	red	strawberry	5
3	blue	prune	2
4	red	cranberry	3

In [63]:
```
color_group = df_fruits.groupby("color")
```

Il est ensuite possible d'extraire des valeurs d'ensemble de ces groupes de données et générer des nouveaux dataframe.

Extraire des données

Combien de fruits a-t-on par couleur ?

In [64]:
```
color_group["color"].count()
```

Out[64]:
```
color
blue       1
red        3
yellow     1
Name: color, dtype: int64
```

Quel est le prix total des fruits par groupe de couleur ?

In [65]:
```
color_group["price"].sum()
```

Out[65]:
```
color
blue        2
red        11
yellow      5
```

Extraire des données

Quel est le prix moyen par couleur ?

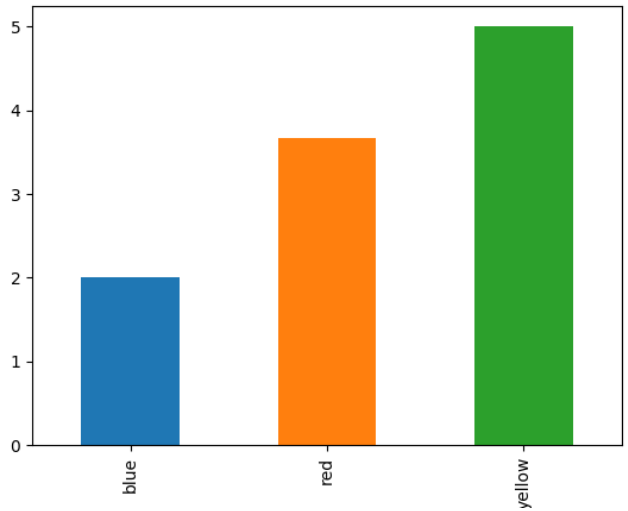
In [66]:
```
1 color_group["price"].mean()
```

Out[66]:
```
color
blue       2.000000
red        3.666667
yellow     5.000000
```

Visualiser les données

Pandas permet aussi de visualiser rapidement les données grâce à la fonction plot intégrée dans le module :

```
1 import matplotlib.pyplot as plt
2 a=color_group["price"].mean()
3 a.plot(kind="bar")
4 plt.show()
```

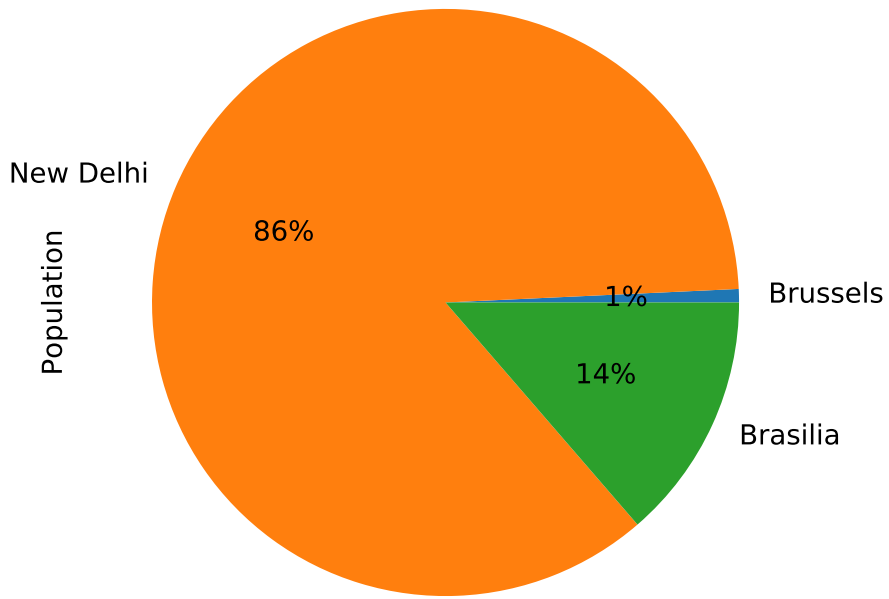


Différents types de représentations graphiques sont disponibles : scatter, hist, box, pie, area...

Visualiser les données

Par exemple en diagramme circulaire :

```
1 import matplotlib.pyplot as plt
2 s1=pd.Series(df['Capital'])
3 s2=pd.Series(df['Population'])
4 s2.plot(kind='pie',autopct='%2.f%%',labels=s1)
5 plt.show()
```



À vos TPs !

1. Ouvrir un terminal :
   - soit sur https ://jupyterhub.ijclab.in2p3.fr
   - soit sur un ordinateur du 336
2. Télécharger la séance d'aujourd'hui :

```
methnum fetch L1/Seance10 TONGROUPE
```

   en remplaçant TONGROUPE par ton numéro de groupe.
3. Sur un ordinateur du bâtiment 336 uniquement, lancer le jupyter :

```
methnum jupyter notebook
```

4. Pour soumettre la séance, dans le terminal taper :

```
methnum submit L1/Seance10 TONGROUPE
```

À vos TPs !

Rappel : votre gitlab personnel sert de sauvegarde pour passer vos documents d'une plateforme à l'autre via les commandes methnum/fetch.