Découverte de Pandas

Xavier Gendre

Pandas

Pandas est un module Python dédié à la manipulation et à l'analyse de données. Il s'agit d'un logiciel libre largement utilisé en Data Science.

L'alias `pd` est généralement utilisé pour le module `pandas`.

```
import pandas as pd
```

- Manipulation de nombreux formats (CSV, Parquet, HDF, …).
- Intégration avec les modules scientifiques comme NumPy et les modules de Machine Learning comme Scikit-learn.
- Outils de visualisation de données.

Series

Pandas offre le type élémentaire `Series` pour représenter un objet *unidimensionnel* similaire à un vecteur et contenant des données de **même type**.

Un objet `Series` peut être créé à partir d'une liste et possède un type `dtype`.

```
s = pd.Series([1, 2, 3])
s
```

```
0    1
1    2
2    3
dtype: int64
```

Par défaut, les indices (`index`) d'un objet `Series` sont des entiers commençant à l'indice `0`.

Il est possible de préciser les indices à la création.

```
s = pd.Series([1, 2, 3], index=["A", "B", "C"])
s
```

```
A    1
B    2
C    3
dtype: int64
```

Cela peut aussi se faire avec un dictionnaire.

```
s = pd.Series({"A": 1, "B": 2, "C": 3})
s
```

```
A    1
B    2
C    3
dtype: int64
```

Index (`index`) et valeurs (`values`) peuvent être récupérés.

```
print(f"Index: {s.index}")
```

```
Index: Index(['A', 'B', 'C'], dtype='object')
```

```
print(f"Valeurs: {s.values}")
```

```
Valeurs: [1 2 3]
```

Un objet `Series` peut également avoir un nom (`name`) pour l'identifier dans un jeu de données.

```
s = pd.Series([1, 2, 3], index=["A", "B", "C"], name="X")
s
```

```
A    1
B    2
C    3
Name: X, dtype: int64
```

L'accès à la valeur d'un élément est possible par sa position ou son index.

```
print(s[1]) # Position
```

```
2
```

```
/tmp/ipykernel_58928/1215828491.py:1: FutureWarning: Series.__getitem__ treating keys as posi
  print(s[1]) # Position
```

```
print(s["B"]) # Index
```

```
2
```

Un sous-objet `Series` s'obtient en passant une liste.

```
print(s[[0, 1]]) # Position (liste)
```

```
A    1
B    2
Name: X, dtype: int64
```

```
/tmp/ipykernel_58928/3637425627.py:1: FutureWarning: Series.__getitem__ treating keys as posi
  print(s[[0, 1]]) # Position (liste)
```

```
print(s[["A", "B"]]) # Index (liste)
```

```
A    1
B    2
Name: X, dtype: int64
```

Les objets `Series` permettent les opérations terme à terme.

```
s0 = pd.Series([1, 2, 3], index=["A", "B", "C"])
s1 = pd.Series([4, 5, 6], index=["A", "B", "C"])
```

```
s0 + s1
```

```
A    5
B    7
C    9
dtype: int64
```

```
s0 * s1
```

```
A     4
B    10
C    18
dtype: int64
```

```
2 * s0
```

```
A    2
B    4
C    6
dtype: int64
```

```
s0**2
```

```
A    1
B    4
C    9
dtype: int64
```

Il faut cependant faire attention aux index de chaque objet.

```
s0 = pd.Series([1, 2, 3], index=["A", "B", "C"])
s1 = pd.Series([4, 5, 6], index=["A", "B", "D"]) # Pas d'indice C

s0 + s1
```

```
A    5.0
B    7.0
C    NaN
D    NaN
dtype: float64
```

Pour le type numérique, des méthodes d'*agrégation* usuelles sont disponibles (`mean`, `var`, `min`, `max`, ...)

```
s0.mean()
```

```
2.0
```

```
s0.max()
```

```
3
```

DataFrame

Un jeu de données n'est généralement pas unidimensionnel et contient plusieurs colonnes de types différents à la façon d'un tableur informatique.

Avec Pandas, ces colonnes sont des `Series` et elles forment un objet `DataFrame`.

Un `DataFrame` est un objet central en Data Science. Les colonnes correspondent généralement à des **variables** et les lignes à des **observations**.

Création d'un DataFrame

Il existe plusieurs façons de créer un `DataFrame` dont l'utilisation d'un dictionnaire de listes :

```
df = pd.DataFrame({
    "Nombre": [1, 2, 3, 4],
    "Label": ["A", "B", "C", "D"],
    "X": [1.2, 3.4, 5.6, 7.8],
    "Bool": [True, False, False, True],
    "Nom": ["Bob", "Ken", "Ben", "Joy"]
})
print(df)
```

```
   Nombre Label    X   Bool  Nom
0       1     A  1.2   True  Bob
1       2     B  3.4  False  Ken
2       3     C  5.6  False  Ben
3       4     D  7.8   True  Joy
```

Un objet `DataFrame` permet un grand nombre de manipulations.

Les opérations suivantes font partie des plus importantes :

- **Sélection** de colonnes/variables,
- **Filtre** sur les lignes,
- **Tri** des lignes,
- **Mutation** pour créer ou modifier des colonnes,
- **Agrégation** pour résumer l'information.

Sélection

Il est possible de manipuler une colonne d'un `DataFrame` comme un objet `Series` par attribut (`df.X`) ou avec la syntaxe des listes (`df["X"]`).

```
print(df.X) # Objet Series
```

```
0    1.2
1    3.4
2    5.6
3    7.8
Name: X, dtype: float64
```

```
print(df["X"]) # Objet Series
```

```
0    1.2
1    3.4
2    5.6
3    7.8
Name: X, dtype: float64
```

Une liste de noms de colonnes permet d'extraire un sous-objet `DataFrame`.

```
print(df[["X", "Nom"]])
```

```
     X  Nom
0  1.2  Bob
1  3.4  Ken
2  5.6  Ben
3  7.8  Joy
```

La méthode `filter` sélectionne les colonnes par nom,

```
print(df.filter(items=["X", "Nom"]))
```

```
     X  Nom
0  1.2  Bob
1  3.4  Ken
2  5.6  Ben
3  7.8  Joy
```

par contenu,

```
print(df.filter(like="om"))
```

```
   Nombre  Nom
0       1  Bob
1       2  Ken
2       3  Ben
3       4  Joy
```

ou par expression régulière.

```
print(df.filter(regex="l$"))
```

```
  Label   Bool
0     A   True
1     B  False
2     C  False
3     D   True
```

Filtre

Filtrer les lignes se fait en passant une `Series` de booléens.

```
print(df[df.X < 5])
```

```
   Nombre Label    X   Bool  Nom
0       1     A  1.2   True  Bob
1       2     B  3.4  False  Ken
```

Pour combiner des conditions, il est possible d'utiliser les opérateurs `&` (**and**), `|` (**or**) et `~` (**not**).

```
print(df[((df.Label == "D") & df.Bool) | ~(df.X >= 5)])
```

```
   Nombre Label    X   Bool  Nom
0       1     A  1.2   True  Bob
1       2     B  3.4  False  Ken
3       4     D  7.8   True  Joy
```

Tri

La méthode `sort_values` permet de trier les lignes.

```
print(df.sort_values(by="Nom"))
```

```
   Nombre Label    X   Bool  Nom
2       3     C  5.6  False  Ben
0       1     A  1.2   True  Bob
3       4     D  7.8   True  Joy
1       2     B  3.4  False  Ken
```

Plusieurs variables peuvent être passées à `sort_values` pour arbitrer les cas d'égalité.

```
df_dalton = pd.DataFrame({
    "Nom": ["Dalton", "Dalton", "Dalton", "Lincoln", "Dalton"],
    "Prenom": ["Joe", "William", "Jack", "Abraham", "Averell"],
    "Taille": [1.4, 1.67, 1.93, 1.93, 2.13]
})

print(df_dalton.sort_values(by=["Nom", "Prenom", "Taille"]))
```

```
       Nom   Prenom  Taille
4   Dalton  Averell    2.13
2   Dalton     Jack    1.93
0   Dalton      Joe    1.40
1   Dalton  William    1.67
3  Lincoln  Abraham    1.93
```

Mutation

Pour créer ou mettre à jour une colonne, il suffit de lui affecter des nouvelles valeurs.

```
df["NEW"] = [42, 43, 44, 45]
print(df)
```

```
   Nombre Label    X   Bool  Nom  NEW
0       1     A  1.2   True  Bob   42
1       2     B  3.4  False  Ken   43
2       3     C  5.6  False  Ben   44
3       4     D  7.8   True  Joy   45
```

```
df.X = df.X + df.Nombre
print(df)
```

```
   Nombre Label     X   Bool  Nom  NEW
0       1     A   2.2   True  Bob   42
1       2     B   5.4  False  Ken   43
2       3     C   8.6  False  Ben   44
3       4     D  11.8   True  Joy   45
```

La création de colonne se fait souvent à l'aide de la méthode `apply` qui permet d'appliquer une fonction sur chaque ligne (`axis=1`) ou sur chaque colonne (`axis=0`, défaut).

```
def f(row):
    return 11 if row.Nom[0] == "B" else 22

df["F"] = df.apply(f, axis=1) # Sur les lignes
print(df)
```

```
   Nombre Label     X   Bool  Nom  NEW   F
0       1     A   2.2   True  Bob   42  11
1       2     B   5.4  False  Ken   43  22
2       3     C   8.6  False  Ben   44  11
3       4     D  11.8   True  Joy   45  22
```

```
print(df[["Nombre", "X", "Bool"]].apply(sum, axis=0)) # Sur les colonnes
```

```
Nombre    10.0
X         28.0
Bool       2.0
dtype: float64
```

Pour réaliser une mutation, il est également possible d'utiliser la méthode `assign`.

```
print(
    df.assign(SUM=df.NEW + df.F)
)
```

```
   Nombre Label     X   Bool  Nom  NEW   F  SUM
0       1     A   2.2   True  Bob   42  11   53
1       2     B   5.4  False  Ken   43  22   65
2       3     C   8.6  False  Ben   44  11   55
3       4     D  11.8   True  Joy   45  22   67
```

Agrégation

La méthode agg permet d'utiliser des *fonctions d'agrégation* sur les colones (ou sur les lignes).

```
print(
    df[["Nombre", "X", "Bool"]].agg(["count", "mean", "var"])
)
```

```
         Nombre          X      Bool
count  4.000000   4.000000  4.000000
mean   2.500000   7.000000  0.500000
var    1.666667  17.066667  0.333333
```

Il est aussi possible d'utiliser des *fonctions d'agrégation* différentes sur chaque colonne.

```python
print(
    df[["Nombre", "X", "F"]].agg(
        {
            "Nombre": ["sum", "min"],
            "X": ["sum", "max"],
            "F": pd.Series.nunique
        }
    )
)
```

```
          Nombre     X    F
sum         10.0  28.0  NaN
min          1.0   NaN  NaN
max          NaN  11.8  NaN
nunique      NaN   NaN  2.0
```

Toute fonction qui s'applique à un objet `Series` peut être utilisée comme agrégateur :

- Pandas dispose des classiques :
  - `"count"`,
  - `"min"` / `"max"`,
  - `"sum"` / `"mean"` / `"median"`,
  - `"var"` / `"std"`, …
- `pd.Series.nunique` compte les valeurs distinctes,
- `pd.unique` retourne la liste des valeurs distinctes,
- fonctions personnalisées, …

Formats standards

En pratique, les jeux de données proviennent de sources extérieures. Pandas permet de lire un grand nombre de formats standards :

- CSV avec `read_csv`,
- Parquet avec `read_parquet`,
- Excel avec `read_excel`,
- HDF avec `read_hdf`,
- …

Le fichier `hflights.csv` contient un jeu de données relatif aux vols partant des aéroports de Houston : IAH (*George Bush Intercontinental*) et HOU (*Houston Hobby*).

```
hflights = pd.read_csv("data/hflights.csv")
hflights.dtypes
```

```
Year                 int64
Month                int64
DayofMonth           int64
DayOfWeek            int64
DepTime            float64
ArrTime            float64
UniqueCarrier       object
FlightNum            int64
TailNum             object
ActualElapsedTime  float64
AirTime            float64
ArrDelay           float64
DepDelay           float64
Origin              object
Dest                object
Distance             int64
TaxiIn             float64
TaxiOut            float64
Cancelled            int64
CancellationCode    object
Diverted             int64
dtype: object
```

La méthode `head` permet d'avoir un aperçu des données chargées.

```
print(hflights.head())
```

```
   Year  Month  DayofMonth  DayOfWeek  DepTime  ArrTime UniqueCarrier  \
0  2011      1           1          6   1400.0   1500.0            AA
1  2011      1           2          7   1401.0   1501.0            AA
```

```
2  2011      1           3           1   1352.0   1502.0            AA
3  2011      1           4           2   1403.0   1513.0            AA
4  2011      1           5           3   1405.0   1507.0            AA

   FlightNum TailNum  ActualElapsedTime  ...  ArrDelay  DepDelay  Origin Dest  \
0        428  N576AA               60.0  ...     -10.0       0.0     IAH  DFW
1        428  N557AA               60.0  ...      -9.0       1.0     IAH  DFW
2        428  N541AA               70.0  ...      -8.0      -8.0     IAH  DFW
3        428  N403AA               70.0  ...       3.0       3.0     IAH  DFW
4        428  N492AA               62.0  ...      -3.0       5.0     IAH  DFW

  Distance  TaxiIn  TaxiOut  Cancelled  CancellationCode Diverted
0      224     7.0     13.0          0               NaN        0
1      224     6.0      9.0          0               NaN        0
2      224     5.0     17.0          0               NaN        0
3      224     9.0     22.0          0               NaN        0
4      224     9.0      9.0          0               NaN        0

[5 rows x 21 columns]
```

Les données peuvent maintenant être manipulées.

La méthode `apply` fonctionne aussi sur les objets `Series` qui forment les colonnes du `DataFrame`.

Les fonctions anonymes (`lambda`) sont souvent utilisées dans ce cadre.

```
carrier_map = {
    "AA": "American",           "AS": "Alaska",     "B6": "JetBlue",
    "CO": "Continental",        "DL": "Delta",      "OO": "SkyWest",
    "UA": "United",             "US": "US_Airways", "WN": "Southwest",
    "EV": "Atlantic_Southeast", "F9": "Frontier",   "FL": "AirTran",
    "MQ": "American_Eagle",     "XE": "ExpressJet", "YV": "Mesa",
}

hflights["UniqueCarrier"] = hflights.UniqueCarrier.apply(
    lambda carrier: carrier_map[carrier]
)
```

Données manquantes

La variable `CancellationCode` contient de nombreuses données manquantes encodées par `nan` du module `numpy` ou par `NA` de `pandas` selon les jeux de données.

```
hflights.CancellationCode.head()
```

```
0    NaN
1    NaN
2    NaN
3    NaN
4    NaN
Name: CancellationCode, dtype: object
```

```
hflights.CancellationCode.unique()
```

```
array([nan, 'A', 'B', 'C', 'D'], dtype=object)
```

La fonction `isna` permet de détecter ces données manquantes (la fonction `notna` fait le contraire).

```
pd.isna(hflights.CancellationCode).head()
```

```
0    True
1    True
2    True
3    True
4    True
Name: CancellationCode, dtype: bool
```

La méthode `dropna` supprime les données manquantes.

```
hflights.CancellationCode.dropna().head()
```

```
194    A
210    B
323    B
335    A
347    B
Name: CancellationCode, dtype: object
```

La méthode `fillna` permet de remplacer ces données manquantes par une valeur. L'argument `inplace` est commun à de nombreuses méthodes et il permet de mettre à jour le contenu du `DataFrame` sans affectation.

```
hflights.CancellationCode.fillna("", inplace=True)
print(hflights.head(2))
```

```
   Year  Month  DayofMonth  DayOfWeek  DepTime  ArrTime UniqueCarrier  \
0  2011      1           1          6   1400.0   1500.0      American
1  2011      1           2          7   1401.0   1501.0      American

   FlightNum TailNum  ActualElapsedTime  ...  ArrDelay  DepDelay  Origin Dest  \
0        428  N576AA               60.0  ...     -10.0       0.0     IAH  DFW
1        428  N557AA               60.0  ...      -9.0       1.0     IAH  DFW

  Distance  TaxiIn  TaxiOut  Cancelled  CancellationCode Diverted
0      224     7.0     13.0          0                         0
1      224     6.0      9.0          0                         0

[2 rows x 21 columns]
```

```
cancel_map = {
    "A": "carrier",
    "B": "weather",
    "C": "national air system",
    "D": "security",
    "": "not cancelled",
}
```

```
hflights["CancellationCode"] = hflights.CancellationCode.apply(
    lambda cancel_code: cancel_map[cancel_code]
)

hflights.CancellationCode.head()
```

```
0    not cancelled
1    not cancelled
2    not cancelled
3    not cancelled
4    not cancelled
Name: CancellationCode, dtype: object
```

Regroupement

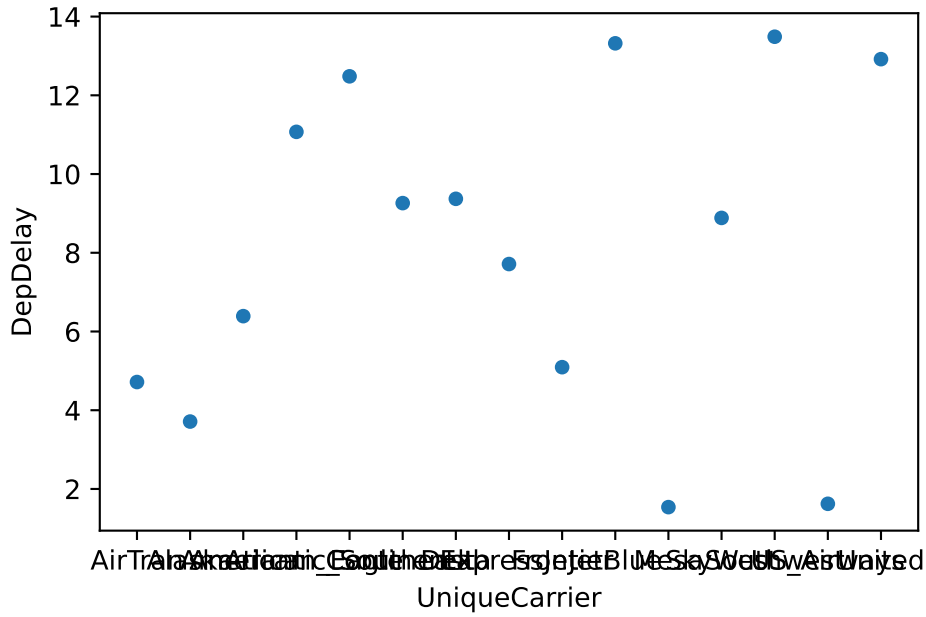
Les données peuvent être groupées avec `groupby` selon des variables. Ceci est très souvent utilisé avec la méthode `agg`.

```
print( # Remarquer la syntaxe de cet exemple
    hflights
    .groupby(hflights.CancellationCode)
    .agg( # NamedAgg permet de nommer les agrégations
        FlightCount=pd.NamedAgg(column="FlightNum", aggfunc="count"),
        DelayMean=pd.NamedAgg(column="DepDelay", aggfunc="mean"),
        DelayStd=pd.NamedAgg(column="DepDelay", aggfunc="std"),
    )
    .sort_values(by="FlightCount", ascending=False)
)
```

```
                     FlightCount  DelayMean    DelayStd
CancellationCode
not cancelled             224523   9.433065   28.759280
weather                     1652  53.615385   58.146764
carrier                     1202  45.250000  105.880133
national air system          118  53.500000   41.719300
security                       1         NaN         NaN
```

Cela devient très utile avec les graphiques

```
(
    hflights.DepDelay
    .groupby(hflights.UniqueCarrier)
    .agg("mean")
    .reset_index() # Index UniqueCarrier en colonne
    .plot.scatter(x="UniqueCarrier", y="DepDelay")
)
```



```
import numpy as np # Hello NumPy :-)
```

```
dep_delay_summary = (
    hflights[
        np.isfinite(hflights.DepDelay) # isfinite de NumPy
        & (hflights.DepDelay > 0)
        & (hflights.CancellationCode != "not cancelled")
    ]
    .groupby( # Groupement sur 2 colonnes
        [hflights.CancellationCode, hflights.UniqueCarrier]
```

```
    )
    .agg(
        count=pd.NamedAgg(column="FlightNum", aggfunc="count"),
        min=pd.NamedAgg(column="DepDelay", aggfunc="min"),
        max=pd.NamedAgg(column="DepDelay", aggfunc="max"),
        med=pd.NamedAgg(column="DepDelay", aggfunc="median"),
    )
)
print(dep_delay_summary.nlargest(2, "count")) # Tri + Filtre
```

```
                                count  min    max   med
CancellationCode UniqueCarrier
weather          ExpressJet        13  1.0  173.0  64.0
carrier          ExpressJet         4  5.0  271.0  91.0
```

Mise en forme

Les jeux de données comme dans l'exemple précédent sont dits au **format long**.

```
print(dep_delay_summary.reset_index())
```

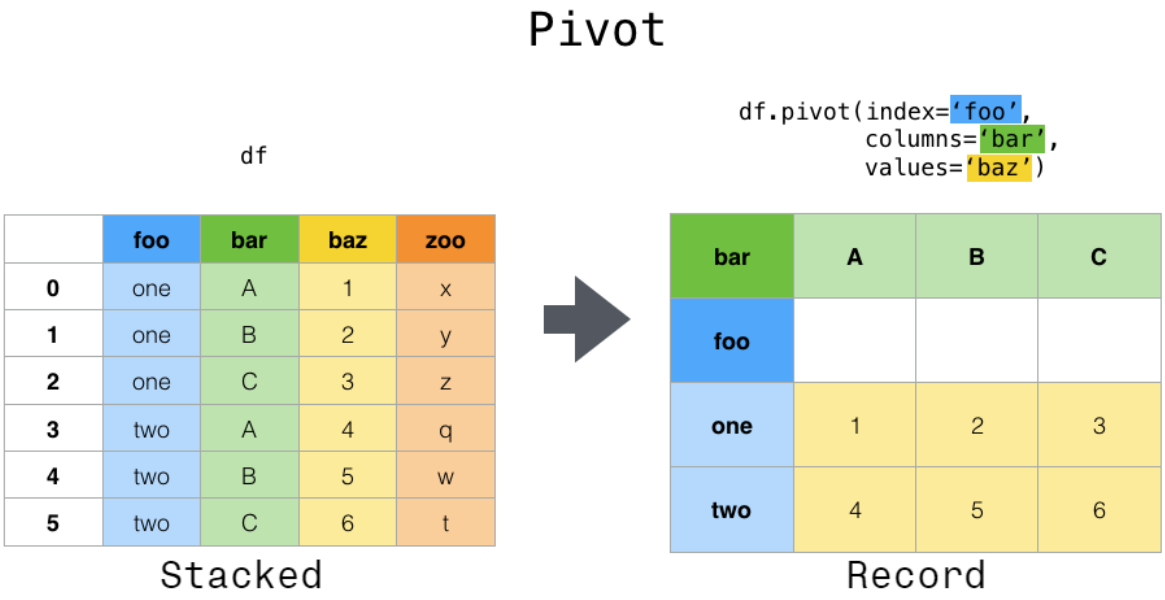
```
       CancellationCode       UniqueCarrier  count    min    max    med
0               carrier             AirTran      1   64.0   64.0   64.0
1               carrier            American      2    3.0    8.0    5.5
2               carrier  Atlantic_Southeast      1  220.0  220.0  220.0
3               carrier         Continental      1  187.0  187.0  187.0
4               carrier               Delta      1   42.0   42.0   42.0
5               carrier          ExpressJet      4    5.0  271.0   91.0
6               carrier             SkyWest      3   27.0   37.0   28.0
7               carrier           Southwest      3    1.0  548.0    3.0
8               carrier          US_Airways      1  153.0  153.0  153.0
9               carrier              United      1  110.0  110.0  110.0
10  national air system         Continental      1   24.0   24.0   24.0
11  national air system          ExpressJet      1   83.0   83.0   83.0
12              weather         Continental      2   26.0  156.0   91.0
13              weather               Delta      1  110.0  110.0  110.0
14              weather          ExpressJet     13    1.0  173.0   64.0
15              weather             SkyWest      3   27.0  103.0   64.0
16              weather          US_Airways      1  135.0  135.0  135.0
```

La méthode `pivot` permet de passer au **format large**.

```
format_large = dep_delay_summary.reset_index().pivot(
    index="CancellationCode",
    columns="UniqueCarrier",
    values="count"
)
print(format_large)
```

```
UniqueCarrier        AirTran  American  Atlantic_Southeast  Continental  \
CancellationCode
carrier                  1.0       2.0                 1.0          1.0
national air system      NaN       NaN                 NaN          1.0
weather                  NaN       NaN                 NaN          2.0

UniqueCarrier        Delta  ExpressJet  SkyWest  Southwest  US_Airways  United
CancellationCode
carrier                1.0         4.0      3.0        3.0         1.0     1.0
national air system    NaN         1.0      NaN        NaN         NaN     NaN
weather                1.0        13.0      3.0        NaN         1.0     NaN
```



	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

La méthode `melt` permet de revenir au **format long**.

```
print(
    format_large.reset_index()
    .melt(id_vars=["CancellationCode"], value_name="count")
)
```

```
       CancellationCode       UniqueCarrier  count
0               carrier             AirTran    1.0
1   national air system             AirTran    NaN
2               weather             AirTran    NaN
3               carrier            American    2.0
4   national air system            American    NaN
5               weather            American    NaN
6               carrier  Atlantic_Southeast    1.0
7   national air system  Atlantic_Southeast    NaN
8               weather  Atlantic_Southeast    NaN
9               carrier         Continental    1.0
10  national air system         Continental    1.0
11              weather         Continental    2.0
12              carrier               Delta    1.0
13  national air system               Delta    NaN
14              weather               Delta    1.0
15              carrier          ExpressJet    4.0
16  national air system          ExpressJet    1.0
17              weather          ExpressJet   13.0
18              carrier             SkyWest    3.0
19  national air system             SkyWest    NaN
20              weather             SkyWest    3.0
21              carrier           Southwest    3.0
22  national air system           Southwest    NaN
23              weather           Southwest    NaN
24              carrier          US_Airways    1.0
25  national air system          US_Airways    NaN
26              weather          US_Airways    1.0
27              carrier              United    1.0
28  national air system              United    NaN
29              weather              United    NaN
```

Il ne reste plus qu'à utiliser `dropna`…

Melt

df3

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

df3.melt(id_vars=['first', 'last'])

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

Les méthodes `stack` et `unstack` jouent un rôle similaire à partir des indices.

```
stacked = dep_delay_summary.stack()
print(stacked)
```

```
CancellationCode  UniqueCarrier
carrier           AirTran        count      1.0
                                 min       64.0
                                 max       64.0
                                 med       64.0
                  American       count      2.0
                                           ...
weather           SkyWest        med       64.0
                  US_Airways     count      1.0
                                 min      135.0
                                 max      135.0
                                 med      135.0
Length: 68, dtype: float64
```

Stack

df2

		A	B
first	second		
bar	one	1	2
	two	3	4
baz	one	5	6
	two	7	8

MultiIndex

stacked = df2.stack()

first	second		
bar	one	A	1
		B	2
	two	A	3
		B	4
baz	one	A	5
		B	6
	two	A	7
		B	8

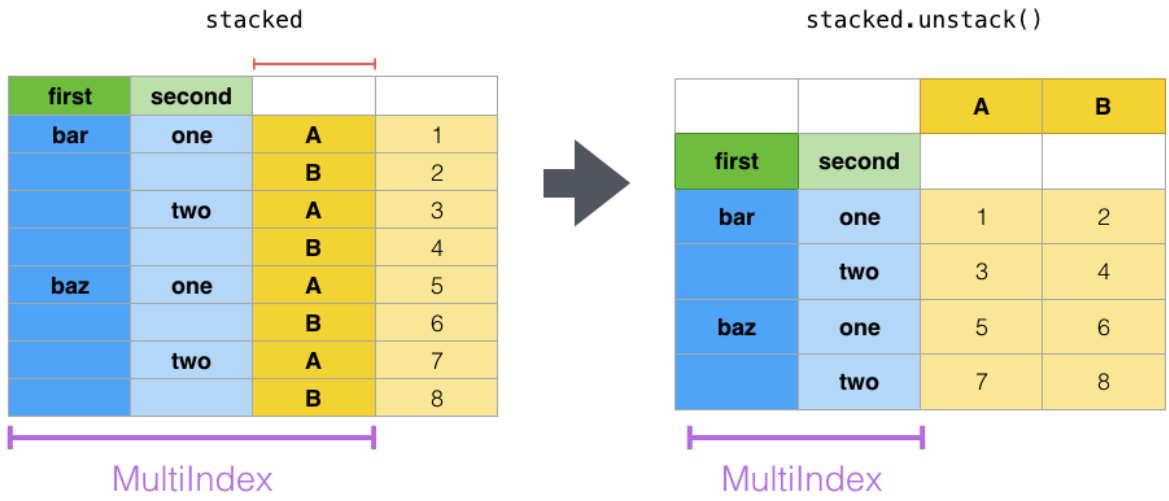
MultiIndex

```
unstacked = stacked.unstack()
print(unstacked)
```

```
                                    count    min    max    med
CancellationCode    UniqueCarrier
carrier             AirTran           1.0   64.0   64.0   64.0
                    American          2.0    3.0    8.0    5.5
                    Atlantic_Southeast  1.0  220.0  220.0  220.0
                    Continental       1.0  187.0  187.0  187.0
                    Delta             1.0   42.0   42.0   42.0
                    ExpressJet        4.0    5.0  271.0   91.0
                    SkyWest           3.0   27.0   37.0   28.0
                    Southwest         3.0    1.0  548.0    3.0
                    US_Airways        1.0  153.0  153.0  153.0
                    United            1.0  110.0  110.0  110.0
national air system Continental       1.0   24.0   24.0   24.0
```

	ExpressJet	1.0	83.0	83.0	83.0
weather	Continental	2.0	26.0	156.0	91.0
	Delta	1.0	110.0	110.0	110.0
	ExpressJet	13.0	1.0	173.0	64.0
	SkyWest	3.0	27.0	103.0	64.0
	US_Airways	1.0	135.0	135.0	135.0

Unstack



À vous de jouer !