9

L'analyse de données avec Pandas

Pandas propose un format de structure de données tabulaires. C'est une bibliothèque qui convient particulièrement au traitement des jeux de données présentés sous forme de tableaux (format csv ou Excel), ou des bases de données relationnelles (comme MySQL ou MongoDB). Pandas (l'abréviation de *panel data*) offre notamment des facilités pour lire, prétraiter, sélectionner, redimensionner, grouper, agréger et visualiser des données.

L'usage est d'importer la bibliothèque Pandas sous l'alias `pd` :

```
>>> import pandas as pd
```

Une présentation complète de Pandas nécessiterait un ouvrage à part entière. Ce chapitre propose une simple introduction des fonctionnalités principales, basée sur l'exemple des communes de France (☞ p. 105, § 7.4). La bibliothèque Pandas lit différents formats de fichiers, le plus simple étant le format csv, c'est-à-dire un fichier dont les colonnes sont séparées par des virgules (*comma separated values*).

9.1. Les bases de Pandas

S'il est bien sûr possible de déchiffrer le fichier à l'aide des fonctions présentées au chapitre 3 ou à l'aide du module Python csv, Pandas propose directement la fonction `pd.read_csv`. Lors de la première lecture d'un fichier, il est recommandé de ne lire que les premières lignes de celui-ci afin de pouvoir raffiner efficacement les options de lecture.

```
villes = pd.read_csv("villes_france.csv", nrows=5)
villes
```

	1	01	ozan	OZAN	ozan.1	Ozan	O250	OSN	01190	284	...
0	2	1	cormoranche-sur-saone	CORMORANCHE-SUR-SAONE	cormoranche sur saone	Cormoranche-sur-Saône	C65652625	KRMRNXSRSN	1290	123	...
1	3	1	plagne-01	PLAGNE	plagne	Plagne	P425	PLKN	1130	298	...
2	4	1	tossiat	TOSSIAT	tossiat	Tossiat	T230	TST	1250	422	...
3	5	1	pouillat	POUILLAT	pouillat	Pouillat	P430	PLT	1250	309	...
4	6	1	torcieu	TORCIEU	torcieu	Torcieu	T620	TRS	1230	421	...

5 rows × 27 columns

L'usage est généralement d'avoir en première ligne d'un fichier csv une ligne d'en-tête

qui explicite le contenu de chacune des colonnes. Pandas fait ici cette hypothèse (à tort) alors que ces informations sont absentes. Les métadonnées relatives aux colonnes sont néanmoins décrites sur la page web du jeu de données. Nous allons n'en sélectionner que quelques-unes : le paramètre `usecols` décrit l'index des colonnes à considérer ; le paramètre `names` les nomme.

```
villes = pd.read_csv(
    "villes_france.csv",
    nrows=5, usecols=[5, 8, 16, 19, 20, 25, 26],
    names=[
        "nom", "code postal", "population", "longitude", "latitude",
        "altitude_min", "altitude_max",
    ],
)
```

	nom	code postal	population	longitude	latitude	altitude_min	altitude_max
0	Ozan	1190	500	4.91667	46.3833	170	205
1	Cormoranche-sur-Saône	1290	1000	4.83333	46.2333	168	211
2	Plagne	1130	100	5.73333	46.1833	560	922
3	Tossiat	1250	1400	5.31667	46.1333	244	501
4	Pouillat	1250	100	5.43333	46.3333	333	770

L'affichage est maintenant plus facile à appréhender : une ligne par entrée, et une colonne par aspect (on parle de *feature*) associé à chaque entrée. Ici, un petit défaut subsiste : les codes postaux ont été interprétés comme des entiers et les zéros initiaux ont alors disparu. Il est possible de spécifier le type associé à chaque colonne dans le paramètre `dtype`.

Une fois les paramètres ajustés, on peut alors lire le fichier en entier après avoir enlevé le paramètre `nrows`. La structure de base de Pandas est le `DataFrame`, un « tableau de données » (`pd.DataFrame`). À l'instar de NumPy, l'attribut `shape` décrit le format du tableau en mémoire, ici 36700 lignes pour 7 colonnes.

```
villes = pd.read_csv(
    "villes_france.csv",
    usecols=[5, 8, 16, 19, 20, 25, 26], names=[
        "nom", "code postal", "population", "longitude", "latitude",
        "altitude_min", "altitude_max",
    ],
    dtype={"code postal": str},
)
```

	nom	code postal	population	longitude	latitude	altitude_min	altitude_max
0	Ozan	01190	500	4.91667	46.3833	170.0	205.0
1	Cormoranche-sur-Saône	01290	1000	4.83333	46.2333	168.0	211.0
2	Plagne	01130	100	5.73333	46.1833	560.0	922.0
3	Tossiat	01250	1400	5.31667	46.1333	244.0	501.0
4	Pouillat	01250	100	5.43333	46.3333	333.0	770.0
...	...	...	...	...	...	...	...
36695	Sada	97640	10195	45.1047	-12.84860	NaN	NaN
36696	Tsingoni	97680	10454	45.1070	-12.78970	NaN	NaN
36697	Saint-Barthélemy	97133	8938	-62.8333	17.91670	NaN	NaN
36698	Saint-Martin	97150	36979	18.0913	-63.08290	NaN	NaN
36699	Saint-Pierre-et-Miquelon	97500	6080	46.7107	1.71819	NaN	NaN

36700 rows × 7 columns

```
>>> type(villes)
pandas.core.frame.DataFrame
>>> villes.shape
(36700, 7)
```

Il est possible d'explorer un DataFrame en n'affichant que les premières/dernières lignes, où en en tirant au hasard dans le fichier.

```
villes.head()  # ou villes.head(10)
villes.tail()
villes.sample(5)  # au hasard
```

	nom	code postal	population	longitude	latitude	altitude_min	altitude_max
7820	Kerbors	22610	300	-3.18333	48.8333	0.0	70.0
28860	Étobon	70400	300	6.68333	47.6500	343.0	585.0
1596	Clumanc	04330	200	6.41667	44.0333	773.0	1703.0
3353	Mesnil-Lettre	10240	100	4.26667	48.4500	121.0	183.0
25526	Voingt	63620	100	2.53333	45.8000	715.0	814.0

Chaque colonne peut être sélectionnée par la notation entre crochets `df["population"]` ou, si la syntaxe qui en résulte le permet, avec la notation pointée `df.population`. Une colonne est une structure `pd.Series`.

```
>>> type(villes.population)
pandas.core.series.Series

>>> villes.population  # équivalent à villes["population"]
0          500
1         1000
2          100
3         1400
4          100
         ...
36695    10195
36696    10454
36697     8938
36698    36979
36699     6080
Name: population, Length: 36700, dtype: int64
```

Une série consiste en un tableau NumPy, accessible par l'attribut `values`, un `dtype`, un index, et, le cas échéant, un nom `name`.

```
>>> (villes.population.values, villes.population.dtype,
...  villes.population.index, villes.population.name)
(array([  500,  1000,   100, ...,  8938, 36979,  6080]),
 dtype('int64'),
 RangeIndex(start=0, stop=36700, step=1),
 'population')
```

On peut indexer un DataFrame à l'aide d'une liste de noms de colonnes pour n'en extraire que certaines. Si la liste n'a qu'un seul élément, Pandas retourne un tableau (`pd.DataFrame`) à une seule colonne, différent d'une colonne (`pd.Series`).

```
villes[["population"]]
```

	population
0	500
1	1000
2	100
3	1400
4	100
...	...
36695	10195
36696	10454
36697	8938
36698	36979
36699	6080

36700 rows × 1 columns

On utilise en général l'indexation par une liste pour sélectionner un jeu de *features* :

```
villes[["nom", "population"]].head()
```

	nom	population
0	Ozan	500
1	Cormoranche-sur-Saône	1000
2	Plagne	100
3	Tossiat	1400
4	Pouillat	100

De nombreuses informations parmi celles présentées ici sont rassemblées dans le résultat de la méthode `.info()`. Celle-ci est en réalité peu utilisée, mais elle rassemble toutes les informations pertinentes quant aux structures de données étudiées. La méthode `.describe()` offre un autre type d'informations statistiques sur la distribution de chacune des *features*.

```
>>> villes.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 36700 entries, 0 to 36699
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   nom           36700 non-null  object
 1   code postal   36700 non-null  object
 2   population    36700 non-null  int64
 3   longitude     36700 non-null  float64
 4   latitude      36700 non-null  float64
 5   altitude_min  36568 non-null  float64
 6   altitude_max  36568 non-null  float64
dtypes: float64(4), int64(1), object(2)
memory usage: 2.0+ MB

>>> villes.describe()
```

	population	longitude	latitude	altitude_min	altitude_max
count	3.670000e+04	36700.000000	36700.000000	36568.000000	36568.000000
mean	1.751080e+03	2.786424	46.691117	193.156831	391.105694
std	1.460775e+04	2.966138	5.751918	194.694801	449.308488
min	0.000000e+00	-62.833300	-63.082900	-5.000000	0.000000
25%	2.000000e+02	0.700000	45.150000	62.000000	140.000000
50%	4.000000e+02	2.650000	47.383300	138.000000	236.000000
75%	1.000000e+03	4.883330	48.833300	253.000000	435.000000
max	2.211000e+06	49.443600	55.697200	1785.000000	4807.000000

9.2. Visualisation, sélection, indexation

Les exemples précédents illustrent comment fonctionnait l'opérateur crochets `[]` sur un `pd.DataFrame` Pandas : une chaîne de caractères en argument renvoie une *feature* de type `pd.Series`, une liste de chaînes de caractères renvoie un sous-tableau de type `pd.DataFrame`.

Il est également possible de procéder à une indexation par ligne. À l'image de NumPy, on peut procéder à une indexation par masque ou par indice. D'une manière générale, cette indexation se fait à l'aide du mot-clé `.loc` :

```
>>> villes.loc[(villes.population > 100_000) & (villes.altitude_min > 400)]
```

	nom	code postal	population	longitude	latitude	altitude_min	altitude_max
16123	Saint-Étienne	42000-42100-42230	172700	4.4	45.4333	422.0	1117.0

La colonne non nommée située le plus à gauche de l'affichage ci-dessus se nomme *index*. Il existe différentes manières d'indexer un `pd.DataFrame`. Un index numérique peut faire l'affaire :

```
>>> villes.index
RangeIndex(start=0, stop=36700, step=1)
>>> villes.loc[16123]
nom                 Saint-Étienne
code postal     42000-42100-42230
population                 172700
longitude                     4.4
latitude                  45.4333
altitude_min                  422
altitude_max                 1117
Name: 16123, dtype: object
```

Il est également possible de choisir une colonne sur laquelle indexer le tableau, par exemple le nom, ou le code postal. Si l'index est unique, un `pd.Series` est renvoyé, sinon on récupère un sous-tableau `pd.DataFrame`.

```
>>> villes.set_index("nom").loc["Cannes"]
code postal     06400-06150
population            72900
longitude           7.01667
latitude              43.55
altitude_min              0
altitude_max            260
Name: Cannes, dtype: object
```

On notera ici que la plupart des opérations Pandas ont un comportement par défaut qui ne modifie pas le `pd.DataFrame` mais en renvoie une copie modifiée. Ce paradigme favorise une expression des traitements de données chaînées, c'est-à-dire où les opérations sont empilées les unes sur les autres de manière linéaire.

```
>>> villes.set_index("nom").loc["Saint-Martin"]
```

nom	code postal	population	longitude	latitude	altitude_min	altitude_max
Saint-Martin	32300	400	0.366667	43.5000	159.0	263.0
Saint-Martin	54450	100	6.752780	48.5681	241.0	301.0
Saint-Martin	65360	400	0.083333	43.1667	332.0	489.0
Saint-Martin	66220	100	2.466670	42.7833	268.0	642.0
Saint-Martin	67220	300	7.300000	48.3500	268.0	615.0
Saint-Martin	83560	200	5.884780	43.5892	343.0	582.0
Saint-Martin	97150	36979	18.091300	-63.0829	NaN	NaN

L'argument `.loc` supporte un deuxième argument pour une sélection à la fois sur les lignes et les colonnes.

```
>>> villes.set_index("code postal").loc["74110", ["nom", "altitude_max"]]
```

code postal	nom	altitude_max
74110	Montriond	2340.0
74110	Morzine	2460.0
74110	Essert-Romand	1780.0
74110	La Côte-d'Arbroz	2240.0

Quand l'index est numérique, certaines opérations peuvent perturber l'ordre des index.

```
>>> villes.sort_values("nom")
```

	nom	code postal	population	longitude	latitude	altitude_min	altitude_max
26263	Aast	64460	200	-0.083333	43.2833	367.0	393.0
21095	Abainville	55130	300	5.500000	48.5333	282.0	388.0
22969	Abancourt	59265	400	3.216670	50.2333	36.0	70.0
23403	Abancourt	60220	700	1.766670	49.7000	170.0	222.0
20841	Abaucourt	54610	300	6.250000	48.9000	182.0	235.0
…	…	…	…	…	…	…	…

36700 rows × 7 columns

```
>>> villes.sort_values("nom").index
Int64Index([26263, 21095, 22969, 23403, 20841, 21196,  8852,  9102, 16793, 9063,
            ...
            11020, 21409,  6232, 21396, 21507, 10973,  7783, 11136, 25430, 1228],
           dtype='int64', length=36700)
```

Dans ce cas, l'argument `.iloc` prend tout son sens : `.loc` procède à une indexation basée sur l'index du `pd.DataFrame` alors que `.iloc` permet de compter les lignes dans l'ordre dans lequel elles apparaissent, que les index aient été modifiés ou qu'une opération ait modifié l'ordre des lignes :

```
>>> villes.set_index("nom").iloc[0]
code postal       01190
population          500
longitude       4.91667
latitude        46.3833
altitude_min        170
altitude_max        205
Name: Ozan, dtype: object

>>> villes.sort_values("altitude_max", ascending=False).iloc[0]
nom             Chamonix-Mont-Blanc
code postal                   74400
population                     9000
longitude                   6.86667
latitude                    45.9167
altitude_min                    995
altitude_max                   4807
Name: 30375, dtype: object
```

Itération sur les lignes d'un tableau. L'itération ligne par ligne est possible avec l'opérateur `.iterrows()`. Elle renvoie des tuples `index, ligne` ; il est toutefois préférable de toujours réfléchir à une manière d'obtenir le résultat voulu à l'aide d'opérations vectorielles, qui sont beaucoup plus efficaces.

```
for index, ligne in villes.set_index("code_postal").iterrows():
    print(index, ligne.nom)
    break
```

```
01190 Ozan
```

```
%%time
# En itérant sur les lignes
sorted((ligne.population, ligne.nom) for index, ligne in villes.iterrows())[-5:]
```

```
CPU times: user 4.86 s, sys: 24.2 ms, total: 4.89 s
Wall time: 5.19 s

[(344900, 'Nice'),
 (439600, 'Toulouse'),
 (474900, 'Lyon'),
 (851400, 'Marseille'),
 (2211000, 'Paris')]
```

```
%%time
# En écriture vectorielle
villes.sort_values("population").tail()[["population", "nom"]]
```

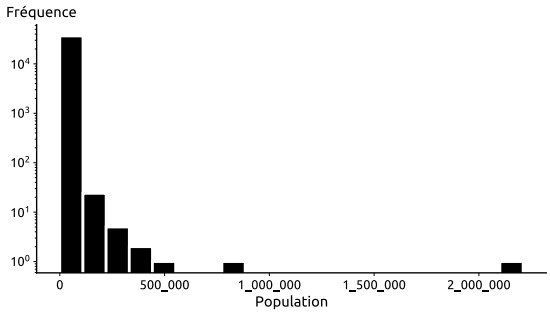
```
CPU times: user 11.7 ms, sys: 1.1 ms, total: 12.8 ms
Wall time: 15.3 ms
```

	population	nom
2049	344900	Nice
11718	439600	Toulouse
28152	474900	Lyon
4439	851400	Marseille
30437	2211000	Paris

Intégration avec Matplotlib. Les `pd.DataFrame` et les `pd.Series` sont tous équipés du mot-clé `.plot` qui donne accès à l'ensemble des méthodes Matplotlib d'affichage. On peut par exemple afficher facilement la distribution que suit une *feature* particulière, ici la population des communes :

```
fig, ax = plt.subplots(figsize=(10, 5))
villes["population"].plot.hist(ax=ax, bins=20, lw=3, ec="w", fc="k")
ax.set_yscale("log")  # axe logarithmique pour explorer la distribution
```

Cette distribution permet alors de choisir judicieusement des critères pour sélectionner certaines lignes de notre tableau. Ici on sélectionne les communes qui ont plus de 200 000 habitants puis on les trie par ordre décroissant de population.




```
villes.loc[villes.population > 200_000, ["nom", "population"]].sort_values(
    "population", ascending=False
).style.format({"population": "{:_}"})
```

	nom	population
30437	Paris	2_211_000
4439	Marseille	851_400
28152	Lyon	474_900
11718	Toulouse	439_600
2049	Nice	344_900
16755	Nantes	283_300
27303	Strasbourg	272_100
13338	Montpellier	253_000
12678	Bordeaux	235_900
22744	Lille	225_800
13467	Rennes	206_700

On notera ici le mot-clé `.style` qui donne accès à un grand nombre de fonctionnalités Pandas pour personnaliser l'affichage d'un `pd.DataFrame` (ici on ajoute un séparateur de milliers sur l'affichage des populations). Cette possibilité offerte par Pandas ne sera pas détaillée dans cet ouvrage mais le lecteur pourra se référer à la documentation officielle (en anglais) : https://pandas.pydata.org/pandas-docs/stable/user_guide/style.html

Les méthodes `eval` et `query`. À l'image de `numexpr`, Pandas met à disposition deux méthodes particulières qui compilent des expressions et les exécutent sur le `pd.DataFrame` en une seule itération.

La méthode `.eval()` évalue l'expression passée en paramètre :

```
>>> # valeur médiane des populations des communes de France
>>> villes.eval("population.median()")
400.0
>>> villes.eval("altitude_max - altitude_min")
0         35.0
1         43.0
2        362.0
3        257.0
         ...
36696      NaN
36697      NaN
36698      NaN
36699      NaN
Length: 36700, dtype: float64
```

C'est surtout la méthode `.query()` qui est couramment utilisée pour sélectionner les lignes d'un `pd.DataFrame` en fonction d'un critère. Ce formalisme simplifie l'écriture (plus de souplesse dans la syntaxe), limite les erreurs (le nom du `pd.DataFrame`, ici `villes` n'a pas besoin d'être rappelé) et améliore la performance du code (une seule itération contre trois dans cet exemple simple).

```
villes.loc[(villes.altitude_min > 1000) & (villes.population > 2000)]
villes.query("altitude_min > 1000 and population > 2000")
```

Si l'expression doit évaluer le contenu d'une variable locale, on peut la rappeler à l'aide du symbole `@` :

```
alt_value, pop_value = 1000, 2000
villes.query("altitude_min > @alt_value and population > @pop_value")
```

	nom	code postal	population	longitude	latitude	altitude_min	altitude_max
1701	Barcelonnette	04400	2700	6.65000	44.3833	1115.0	2680.0
1925	Briançon	05100	11600	6.65000	44.9000	1167.0	2540.0
30043	Modane	73500	3800	6.66667	45.2000	1054.0	3560.0
30140	Tignes	73320	2200	6.91667	45.5000	1440.0	3747.0
30182	Megève	74120	3900	6.61667	45.8667	1027.0	2485.0

9.3. Enrichissement, agrégation

Au-delà des fonctionnalités de visualisation et de sélection, Pandas permet également de modifier et d'enrichir les structures `pd.DataFrame` et `pd.Series`.

Il est notamment possible de renommer des colonnes. C'est le choix que nous faisons dans l'exemple qui nous occupe : pour pouvoir bénéficier de la notation pointée sur les codes postaux, on remplace l'espace par un caractère _. Comme la plupart des fonctionnalités Pandas, celle-ci renvoie de nouvelles structures de données sans modifier les structures d'origine : cette particularité permet notamment de chaîner du code (☞ p. 165, § 12). Si on souhaite enregistrer la modification, on peut remplacer la variable d'origine.

```
villes = villes.rename(columns={"code postal": "code_postal"})
```

Supposons que l'on souhaite agréger les données qui nous sont fournies par département. Il est possible de reconstruire le département à partir des deux premiers chiffres du code postal. La méthode `.apply()` prend en paramètre une fonction, anonyme ou non, à appliquer à chacun des éléments de la `pd.Series`.

```
villes.code_postal.apply(lambda code: code[:2])
```

Pour certains types de données, notamment les chaînes de caractères `str` et les données temporelles, un attribut permet de propager les méthodes associées pour les appliquer à chacun des éléments de la `pd.Series`. Ainsi, pour obtenir le même résultat, on peut appliquer l'opérateur `[:2]` à l'attribut `.str` :

```
>>> villes.code_postal.str[:2]

0        01
1        01
2        01
3        01
4        01
```

```
         ..
36695    97
36696    97
36697    97
36698    97
36699    97
Name: code_postal, Length: 36700, dtype: object
```

Toutes les méthodes applicables aux chaînes de caractères sont disponibles, par exemple `.str.lower()` ou `.str.find("0")`. Les méthodes applicables aux données temporelles seront utilisées dans un exemple plus loin (☞ p. 241, § **16.1**) : elles sont appliquées à l'attribut `.dt`, comme `.dt.day`, `.dt.total_seconds()` ou `.dt.tz_localize()`.

La série étant toujours très longue, on peut agréger cette `pd.Series` pour n'afficher que les éléments uniques. Un tri de la série préalable permet de récupérer les éléments uniques dans l'ordre lexicographique :

```
>>> villes.code_postal.str[:2].sort_values().unique()
```

```
array(['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11',
       '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22',
       '23', '24', '25', '26', '27', '28', '29', '30', '31', '32', '33',
       '34', '35', '36', '37', '38', '39', '40', '41', '42', '43', '44',
       '45', '46', '47', '48', '49', '50', '51', '52', '53', '54', '55',
       '56', '57', '58', '59', '60', '61', '62', '63', '64', '65', '66',
       '67', '68', '69', '70', '71', '72', '73', '74', '75', '76', '77',
       '78', '79', '80', '81', '82', '83', '84', '85', '86', '87', '88',
       '89', '90', '91', '92', '93', '94', '95', '97'], dtype=object)
```

Avant d'assigner le département à chaque commune, il conviendra de traiter deux cas particuliers :

— les codes postaux de Corse commencent par 200 ou 201 pour le département 2A (Corse-du-Sud) et par 202 ou 206 pour le département 2B (Haute-Corse) ;
— les départements d'outre-mer s'écrivent sur trois chiffres qui commencent par 97.

Commençons par le plus simple, on peut créer un vecteur qui traite le cas particulier des DOM à l'aide d'un branchement `np.where(condition, valeur_si_vrai, valeur_si_faux)`, puis ajouter une colonne `departement` à l'aide de la méthode `.assign()` :

```
villes = villes.assign(
    departement=np.where(
        villes.code_postal.str.startswith("97"),
        villes.code_postal.str[:3],
        villes.code_postal.str[:2],
    )
)
```

Le cas particulier de la Corse nous permet d'illustrer une manière de modifier le contenu d'un `pd.DataFrame` sans retourner de copie. Si cette manière de procéder manque d'élégance, il conviendra néanmoins d'y songer quand elle clarifie la lisibilité du code :

```
# On utilise ici .contains qui permet l'utilisation d'expressions régulières
villes.loc[villes.code_postal.str.contains("^20[01]"), "departement"] = "2A"
villes.loc[villes.code_postal.str.contains("^20[26]"), "departement"] = "2B"
```

Il convient alors de confirmer le résultat :

```
array(['01', '02', '03', ...
       '12', '13', '14', '15', '16', '17', '18', '19', '21', '22', '23',
       '24', '25', '26', '27', '28', '29', '2A', '2B', ...
       '88', '89', '90', '91', '92', '93', '94', '95', '971', '972',
       '973', '974', '975', '976'], dtype=object)
```

L'ajout de cette colonne nous permet alors de procéder à des agrégations par département. La méthode qui permet ces opérations est `.groupby()`. Appelée seule, elle ne renvoie qu'un objet de type `DataFrameGroupBy` sans grand intérêt. Cette structure permet néanmoins d'appliquer des opérations d'agrégation.

```
>>> villes.groupby("departement")
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fda22877520>
```

Pour mieux appréhender l'opérateur, il est possible d'itérer dessus : Pandas renvoie alors une valeur unique de clé (ici `departement`), puis le sous-tableau de villes pour lequel toutes les valeurs de `departement` sont égales à la clé :

```
>>> for dept, df in villes.groupby("departement"):
...     print(f"Clé: {dept}; taille: {df.shape}")
Clé: 01; taille: (424, 8)
Clé: 02; taille: (816, 8)
Clé: 03; taille: (319, 8)
Clé: 04; taille: (193, 8)
Clé: 05; taille: (182, 8)
[tronqué]
```

L'agrégation est alors accessible suivant différentes approches :

— la même fonction d'agrégation suivant toutes les *features* (la valeur médiane dans l'exemple ci-dessous, réduite automatiquement aux seules *features* numériques) ;

```
villes.groupby("departement").median()
```

departement	population	longitude	latitude	altitude_min	altitude_max
01	700.0	5.350000	46.10000	237.0	425.0
02	300.0	3.500000	49.55000	72.0	166.0
03	400.0	3.200000	46.33330	250.0	372.0
…	…	…	…	…	…
974	26186.0	-20.979550	55.33470	NaN	NaN
975	6080.0	46.710700	1.71819	NaN	NaN
976	9834.0	45.120000	-12.79820	NaN	NaN

102 rows × 5 columns

— une fonction (ou liste de fonctions) d'agrégation à appliquer à chacune des *features*. Les fonctions d'agrégation les plus communes sont accessibles par une chaîne de caractères, mais il serait également possible de passer une fonction personnalisée ;

```
stats = villes.groupby("departement").agg(
    dict(
        nom="count",  # nombre de villes
        population="sum",  # population totale
```

```
        longitude="median",  # centre géométrique
        latitude="median",
        altitude_min="min",
        altitude_max="max",
    )
)
```

departement	nom	population	longitude	latitude	altitude_min	altitude_max
01	424	584200	5.350000	46.10000	163.0	1704.0
02	816	540200	3.500000	49.55000	36.0	295.0
03	319	342700	3.200000	46.33330	158.0	1280.0
…	…	…	…	…	…	…
974	24	821136	-20.979550	55.33470	NaN	NaN
975	1	6080	46.710700	1.71819	NaN	NaN
976	17	212645	45.120000	-12.79820	NaN	NaN

102 rows × 5 columns

On peut alors récupérer les départements les plus peuplés par exemple :

```
stats.sort_values("population", ascending=False).head(5)
```

departement	nom	population	longitude	latitude	altitude_min	altitude_max
59	646	2563000	3.26667	50.3500	0.0	271.0
75	1	2211000	2.34445	48.8600	0.0	0.0
13	120	1965400	5.25000	43.5333	0.0	1054.0
69	292	1688600	4.65000	45.8500	140.0	1008.0
92	36	1549600	2.26667	48.8333	21.0	179.0

— la dernière possibilité est d'appliquer une fonction personnalisée à chaque sous-tableau renvoyé, puis de réduire le résultat en un unique tableau. Par exemple si on veut récupérer les deux villes les plus peuplées de chaque département (qui n'incluent pas nécessairement la préfecture) :

```
villes.groupby("departement",).apply(
    # trier chaque tableau par population et garder les deux premières lignes
    lambda df: df.sort_values("population", ascending=False).head(2)
).head(12)[["nom", "population"]]
```

departement		nom	population
01	**375**	Bourg-en-Bresse	40200
	275	Oyonnax	23100
02	**1132**	Saint-Quentin	56800
	478	Soissons	28500
03	**1294**	Montluçon	39500
	1369	Vichy	25200
04	**1717**	Manosque	22300
	1738	Digne-les-Bains	17300
05	**1818**	Gap	38600
	1925	Briançon	11600
06	**2049**	Nice	344900
	1999	Antibes	77000

9.4. Fusion de données

L'inconvénient de notre tableau est qu'il ne contient pas les noms des départements auxquels il fait référence : ceux-ci sont absents du fichier d'origine. Pandas propose des méthodes de fusion de données, ou *jointures*, issues de la théorie des bases de données. Pour bien démarrer, il convient de récupérer un fichier qui associe un code de département à son nom :

```
url = "https://www.data.gouv.fr/fr/datasets/r/70cef74f-70b1-495a-8500-c089229c0254"
departements = pd.read_csv(url)  # Pandas télécharge directement depuis Internet
```

	code_departement	nom_departement	code_region	nom_region
0	01	Ain	84	Auvergne-Rhône-Alpes
1	02	Aisne	32	Hauts-de-France
2	03	Allier	84	Auvergne-Rhône-Alpes
...	...	...	...	...
98	973	Guyane	3	Guyane
99	974	La Réunion	4	La Réunion
100	976	Mayotte	6	Mayotte

101 rows × 4 columns

Pour fusionner deux tables (opération de *jointure* dans le langage des bases de données) à l'aide de la méthode `.merge()`, il faut préciser suivant sur quelle(s) colonne(s) (quelle *clé*) baser notre fusion :

- si les colonnes ont le même nom dans les deux tables, on peut utiliser l'argument `on=` ; sinon, on peut raffiner à l'aide de `left_on=` et `right_on=` (pour gauche et droite) ;
- si la jointure doit se faire sur l'index, préciser `left_index=True` ou `right_index=True` ;
- la méthode de jointure par défaut est `"inner"`, ce qui signifie que seuls les éléments clés présents dans les deux tables sont conservés. Les autres méthodes sont `"left"` (on conserve tous les éléments de la table de gauche), `"right"` (tous les éléments de la table de droite), `"outer"` (tous les éléments présents dans une table ou l'autre).

Ici, le code du département est la clé du tableau `stats`. Dans le tableau de référence téléchargé sur `https://www.data.gouv.fr`, c'est la colonne `code_departement`.

```
stats_avec_nom = stats.merge(departements, left_index=True, right_on="code_departement")
```

Les résultats précédents deviennent alors plus lisibles :

```
features = [
    "code_departement", "nom_departement", "population", "altitude_min", "altitude_max"
]

# Les 5 départements les plus élevés en altitude
stats_avec_nom.sort_values("altitude_max", ascending=False)[features].head(5)
```

	code_departement	nom_departement	population	altitude_min	altitude_max
74	74	Haute-Savoie	715200	250.0	4807.0
4	05	Hautes-Alpes	134800	460.0	4099.0
38	38	Isère	1188100	134.0	4008.0
73	73	Savoie	409800	207.0	3855.0
3	04	Alpes-de-Haute-Provence	156800	256.0	3410.0

```
# Les 5 départements les plus peuplés
stats_avec_nom.sort_values("population", ascending=False)[features].head(5)
```

	code_departement	nom_departement	population	altitude_min	altitude_max
59	59	Nord	2563000	0.0	271.0
75	75	Paris	2211000	0.0	0.0
12	13	Bouches-du-Rhône	1965400	0.0	1054.0
69	69	Rhône	1688600	140.0	1008.0
92	92	Hauts-de-Seine	1549600	21.0	179.0

9.5. Formats d'échange

Nous n'avons travaillé ici qu'avec le format CSV pour lire des fichiers. Pandas propose de lire et d'écrire depuis plusieurs formats de fichiers. D'une manière générale, le choix du bon format d'échange dépendra de plusieurs questions : est-il nécessaire de distribuer les données ? est-il nécessaire de les lire/écrire rapidement ? les données doivent-elles être lisibles encore longtemps ?

— Le format **CSV** (*comma separated values*) est un format standard et bien connu. La seule nuance qui puisse exister est celle du séparateur (l'option `sep=`) : historiquement la virgule sépare les colonnes, mais dans le monde francophone on utilise souvent le point-virgule. C'est un format facile à décoder mais qui passe mal à l'échelle : quand les fichiers deviennent grands, le décodage devient long et gourmand en mémoire. Aussi, le format ne contient aucune information de type (chaînes de caractères, entiers, etc.) : il faut alors les ajuster manuellement.
D'une manière générale, la bonne pratique veut qu'on ne lise les fichiers CSV qu'une fois et qu'on utilise un autre format s'il est nécessaire de les stocker pour une utilisation future.

— Le format **JSON** (*JavaScript Object Notation*) est un autre format textuel léger, lisible par les humains, mais également lent à décoder. L'avantage par rapport à un fichier CSV est qu'il est possible de distinguer les booléens, les valeurs numériques et les chaînes de caractères dans le fichier.

— Le format ***pickle*** est le format standard de sérialisation Python (☞ p. 41, § 3.4). La représentation binaire des données est simplement écrite dans un fichier. La lecture et l'écriture de ces fichiers sont rapides, et le format garantit de récupérer les données telles quelles après avoir redémarrer l'interpréteur Python. L'inconvénient est que le format de sérialisation peut changer avec les versions de Python et de Pandas. Ce n'est pas un bon format pour partager ou stocker des données à long terme.

— Le format **HDF** (*Hierarchical Data Format*) est un format standard, indépendant de la plateforme et du langage de programmation, efficace pour stocker de gros volumes de données. Il peut y avoir besoin de dépendances supplémentaires pour lire et écrire dans ce format.

— Le format **Apache Parquet** est un format de stockage en colonne, indépendant de la plateforme et du langage de programmation. Le format est bien intégré à Pandas, les opérations de lecture et d'écriture sont rapides et les fichiers produits sont plutôt compacts. Les types de base sont respectés mais certaines structures Python pourraient ne pas être directement exportables. Il peut y avoir besoin de dépendances supplémentaires pour lire et écrire dans ce format.

9.6. Le passage à Pandas 2.0

La principale évolution de Pandas 2.0 concerne l'amélioration de la performance. Pandas a d'abord été développé autour des structures de données NumPy. PyArrow est une bibliothèque Python adaptée pour les grands jeux de données, qui utilise les structures de données Arrow et qui s'intègre très bien avec d'autres systèmes de gestion de gros volumes de données comme Spark ou Parquet.

Il est possible de créer des dataframes Pandas qui supportent directement le format `pyarrow` avec l'option suivante :

```
pd.read_csv(mon_fichier, dtype_backend='pyarrow')
```

Il est également possible de convertir un dataframe existant :

```
stats_pyarrow = stats.convert_dtypes(dtype_backend="pyarrow")
stats_pyarrow
```

Si on compare les deux formats de types, on observe les valeurs `NaN` (du standard IEEE de définition des flottants, également présents dans NumPy) alors que dans la nouvelle version, il deviennent des objets vides étiquetés `<NA>`. Alors que NumPy ne permet pas d'avoir des types de tableau `int64` avec des valeurs vides, Arrow le permet. Ainsi, on peut comparer les `dtypes` des colonnes avec le backend NumPy original `"numpy_nullable"` et avec le backend PyArrow, qui autorise des valeurs nulles parmi les valeurs entières d'altitudes.

```
>>> stats.dtypes
nom               int64
population        int64
longitude       float64
latitude        float64
altitude_min    float64
altitude_max    float64
dtype: object
>>> stats_arrow.dtypes
nom              int64[pyarrow]
population       int64[pyarrow]
longitude       double[pyarrow]
latitude        double[pyarrow]
altitude_min     int64[pyarrow]
altitude_max     int64[pyarrow]
dtype: object
```

9.7. Pandas ou Polars

Polars est une bibliothèque plus récente que Pandas, écrite en Rust, avec un moteur d'exécution qui n'est pas basé sur NumPy et qui est très efficace d'un point de vue de la performance. Certains benchmarks annoncent des performances jusqu'à 30 fois plus rapides.

La syntaxe est légèrement différente de celle de Pandas néanmoins et un code écrit avec Pandas n'est pas exécutable tel quel sur une structure Polars.

Les principales différences sont les suivantes.

```
import polars as pl

villes_pl = pl.from_dataframe(villes.convert_dtypes(dtype_backend="pyarrow"))
villes_pl
```

— La fonction `.loc` n'existe pas :

```
villes.loc[:,'population']
# devient avec polars
villes_pl.select('population')
```

— Le filtre se fait alors avec la fonction `filter` :

```
villes.loc[(villes.altitude_min > 1000) & (villes.population > 2000)]
# devient avec polars
villes_pl.filter((pl.col("altitude_min") > 1000) & (pl.col("population") > 2000))
```

— Les méthodes `.query()` et `.eval()` qui utilisent numexpr ne sont plus pertinentes en Polars, puis les expressions qui utilisent `pl.col("colonne") > valeur` sont de type `pl.Expr` et n'évaluent rien de manière gloutonne sur le dataframe.
— La syntaxe du groupby est aussi légèrement différente :

```
villes.groupby('departement').agg(dict(population="sum"))
# devient avec polars
villes_pl.group_by("departement").agg(pl.sum("population"))
```

— Si on souhaite aggréger la même colonne plusieurs fois, il convient de renommer la colonne avec l'opérateur `.alias` (Polars ne procède pas par multi-index)

```
villes.groupby('departement').agg(dict(population=["sum", "min"]))
# devient avec polars
villes_pl.group_by("departement").agg(
    pl.sum("population"),
    pl.min("population").alias("population_min"),
)
```

— Enfin, l'opérateur `assign` devient `with_columns`, avec également l'utilisation de `.alias` qui est encouragée :

```
villes.assign(
    departement=np.where(
        villes.code_postal.str.startswith("97"),
        villes.code_postal.str[:3],
        villes.code_postal.str[:2],
    )
)
# devient avec polars
villes_pl.with_columns(
    pl.when(pl.col("code_postal").str.starts_with("97"))
    .then(pl.col("code_postal").str.slice(0, 3))
    .otherwise(pl.col("code_postal").str.slice(0, 2))
    .alias("departement")
)
```

Si la manière de penser le code est différente, la documentation, la complétion automatique, voire même les capacités de traduction de code d'outils comme ChatGPT peuvent aider à faire la transition entre Pandas et Polars. On notera que Polars limite le besoin d'utiliser des fonctions anonymes (avec `lambda`) grâce à sa manière de penser l'ajout de nouvelles colonnes avec `.with_columns` ou `.agg`.

Pour l'exemple de code Pandas un peu complexe qui apparaît plus haut :

```
villes.groupby("departement",).apply(
    # trier chaque tableau par population et garder les deux premières lignes
    lambda df: df.sort_values("population", ascending=False).head(2)
).head(12)[['nom', 'population']]
```

on pourra proposer par exemple le code suivant :

```
villes_pl.group_by("departement").agg(
    pl.col("nom", "population").gather(pl.arg_sort_by("population", descending=True)).head(2)
).sort("departement").explode("nom", "population").head(12)
```

Enfin, la documentation de Polars recommande l'utilisation de la fonction `scan_csv` à la place de `read_csv` car cette fonction est évaluée de manière paresseuse et ne lit le fichier en entier que lorsqu'il devient nécessaire.

Sur des tests de performance effectués par la communauté, la version de Pandas basée sur PyArrow peut être jusqu'à 12 fois plus rapide que cette basée sur NumPy, et approcher ainsi les performances de Polars, qui reste toujours plus performant. Les tests incluent généralement une comparaison avec DuckDB (☞ p. 317, § **21.3.1**), un système de traitement de données analytiques similaires à Pandas et Polars, utilisable de manière indépendante (le langage est compatible avec la syntaxe SQL), ou *via* d'autres langages comme Python.