Manipulation de données

Pandas avancé

Xavier Gendre

Mutation partielle

La méthode de mutation présentée dans la première partie modifie une colonne entière. En particulier, elle ne permet pas de limiter *facilement* la mutation à certaines données.

```
df = pd.DataFrame({
    "player_id": [10, 20, 30],
    "player_name": ["Bob", "Joy", "Ben"],
    "player_gender": ["M", "F", "M"],
})

df["player_label"] = "joueur " + df.player_name

print(df)
```

```
   player_id player_name player_gender player_label
0         10         Bob             M   joueur Bob
1         20         Joy             F   joueur Joy
2         30         Ben             M   joueur Ben
```

La méthode `at` permet d'accéder et de modifier une unique entrée d'un `DataFrame` à partir de son indice et de sa colonne.

```
df.at[1, "player_label"]
```

```
'joueur Joy'
```

```
df.at[1, "player_label"] = "Joueuse Joy"

print(df)
```

```
   player_id player_name player_gender player_label
0         10         Bob             M   joueur Bob
1         20         Joy             F  Joueuse Joy
2         30         Ben             M   joueur Ben
```

Pour modifier plusieurs entrées (généralement obtenue par un filtre), l'opération est plus délicate.

```
1 df_male = df[df.player_gender == "M"]
2 df_male.player_label = "Joueur " + df_male.player_name
```

```
/tmp/ipykernel_51226/138216305.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_male.player_label = "Joueur " + df_male.player_name
```

Le `DataFrame` initial n'a pas été modifié par cette opération.

```
1 print(df)
```

```
   player_id player_name player_gender player_label
0         10         Bob             M   joueur Bob
1         20         Joy             F  Joueuse Joy
2         30         Ben             M   joueur Ben
```

Une *copie* `df_male` du `DataFrame` filtré a été créée et a été modifiée par cette opération.

```
1 print(df_male)
```

```
   player_id player_name player_gender player_label
0         10         Bob             M   Joueur Bob
2         30         Ben             M   Joueur Ben
```

Pandas ne peut pas décider ce qui doit être modifié et affiche l'avertissement précédent.

Comme indiqué dans le message d'avertissement, il faut utiliser la méthode `loc` afin de modifier le `DataFrame` initial.

```
df.loc[df.player_gender == "M", "player_label"] = (
    "Joueur " + df[df.player_gender == "M"].player_name
)
print(df)
```

```
   player_id player_name player_gender player_label
0         10         Bob             M   Joueur Bob
1         20         Joy             F  Joueuse Joy
2         30         Ben             M   Joueur Ben
```

Pour ne modifier que la copie sans avertissement, il faut utiliser la méthode `copy`.

```
df_male = df[df.player_gender == "M"].copy()
df_male.player_label = "Joueur " + df_male.player_name
```

Concaténation

La fonction `concat` permet de concaténer des objets `DataFrame` dans une nouvelle copie.

```
df0 = pd.DataFrame(
    {"player_id": [10, 20], "player_name": ["Bob", "Joy"]}
)
df1 = pd.DataFrame(
    {"player_id": [30, 40], "player_name": ["Ken", "Ted"]}
)

new_df = pd.concat([df0, df1])

print(new_df)
```

```
   player_id player_name
0         10         Bob
1         20         Joy
0         30         Ken
1         40         Ted
```

L'argument `ignore_index` permet d'éviter la duplication des indices.

```
new_df = pd.concat([df0, df1], ignore_index=True)

print(new_df)
```

```
   player_id player_name
0         10         Bob
1         20         Joy
2         30         Ken
3         40         Ted
```

Par défaut, des valeurs manquantes sont introduites si les colonnes ne correspondent pas.

```
1 df2 = pd.DataFrame({"player_id": [50, 60]})
2 print(pd.concat([df0, df2]))
```

```
   player_id player_name
0         10         Bob
1         20         Joy
0         50         NaN
1         60         NaN
```

L'argument `join` permet de limiter la concaténation aux colonnes communes.

```
1 print(pd.concat([df0, df2], join="inner"))
```

```
   player_id
0         10
1         20
0         50
1         60
```

Jointure

Une **jointure** est une opération qui permet d'associer deux objets `DataFrame` selon une logique donnée pour créer un nouvel objet contenant les données jointes.

Il est courant de désigner le premier `DataFrame` comme celui de gauche (*left*) et le second comme celui de droite (*right*). La logique de jointure est définie par une ou plusieurs **clés**.

Pour réaliser cette opération, Pandas dispose de la fonction `merge` (et de la fonction `join` qui produit des résultats équivalents).

Clé primaire

Pour faire correspondre les lignes de deux `DataFrame`, il faut trouver une combinaison de variables :

- existante dans les deux `DataFrame`,
- permettant d'identifier de façon unique une ligne.

Une telle combinaison est appelée **clé primaire**.

	prenom	nom	hasard
0	Émile	Zola	1.1234
1	Émile	Basly	0.9876
2	Étienne	Lantier	0.8765
3	Toussaint	Maheu	1.2345
4	Zacharie	Maheu	0.9786

Le couple de variables `prenom` et `nom` permet l'identification unique d'une ligne. Il en va de même pour la variable `hasard`.

Exemple de jointure

```
 1  df_nom = pd.DataFrame(
 2      {
 3          "player_id": [10, 20, 30],
 4          "player_name": ["Bob", "Joy", "Ken"],
 5      }
 6  )
 7  df_score = pd.DataFrame(
 8      {
 9          "player_id": [10, 20, 30],
10          "player_score": [170881, 314159, 424242],
11      }
12  )
13
14  print(
15      pd.merge(df_nom, df_score, on="player_id")
16  )
```

```
   player_id player_name  player_score
0         10         Bob        170881
1         20         Joy        314159
2         30         Ken        424242
```

Un objet `DataFrame` a aussi d'une méthode `merge` qui met en évidence la table de gauche (*left*) et celle de droite (*right*).

```
1 print(df_nom.merge(df_score, on="player_id"))
```

```
   player_id player_name  player_score
0         10         Bob        170881
1         20         Joy        314159
2         30         Ken        424242
```

En dehors de la table de gauche, les paramètres sont identiques, voici les principaux :

- `how` : type de jointure,
- `on` : clé primaire (noms identiques),
- `left_on`/`right_on` : clés avec des noms différents,
- `suffixes` : pour les conflits de noms (`_x` et `_y` par défaut).

Left join

Pour une jointure à gauche (*left join*), la table de gauche est conservée intégralement et elle est complétée par la table de droite dont les données ne sont prises en compte que si la clé primaire correspond.

```
df0 = pd.DataFrame({
    "key": ["K0", "K1", "K2"], "value": [1.1, 2.2, 3.3],
})
df1 = pd.DataFrame({
    "key": ["K1", "K2", "K3"], "value": ["AAA", "BBB", "CCC"],
})

print(df0.merge(df1, how="left", on="key")) # Left join
```

```
  key  value_x value_y
0  K0      1.1     NaN
1  K1      2.2     AAA
2  K2      3.3     BBB
```

Right join

Une jointure à droite (*right join*) est l'opération symétrique où la table de droite est conservée intégralement et elle est complétée par la table de gauche si la clé primaire correspond.

```
print(df0.merge(df1, how="right", on="key")) # Right join
```

```
  key  value_x value_y
0  K1      2.2     AAA
1  K2      3.3     BBB
2  K3      NaN     CCC
```

Inner join

Par défaut, Pandas fait une jointure intérieure (*inner join*) qui ne conserve les données des tables de gauche et de droite que si la clé primaire correspond des deux côtés.

```
1 print(df0.merge(df1, how="inner", on="key")) # Inner join
```

```
  key  value_x value_y
0  K1      2.2     AAA
1  K2      3.3     BBB
```

```
1 print(df0.merge(df1, on="key")) # Comportement par défaut
```

```
  key  value_x value_y
0  K1      2.2     AAA
1  K2      3.3     BBB
```

Outer join (full join)

Enfin, la jointure extérieure (*outer join*) ou jointure complète (*full join*) conserve les données des deux tables dès que la clé primaire correspond au moins d'un côté.

```
1 print(df0.merge(df1, how="outer", on="key")) # Outer join
```

```
  key  value_x value_y
0  K0      1.1     NaN
1  K1      2.2     AAA
2  K2      3.3     BBB
3  K3      NaN     CCC
```

Cross join

La jointure croisée (*cross join*) est un cas particulier sans clé primaire avec tous les appariements possibles entre les deux tables.

```
1 print(df0.merge(df1, how="cross")) # Cross join
```

```
  key_x  value_x key_y value_y
0    K0      1.1    K1      AAA
1    K0      1.1    K2      BBB
2    K0      1.1    K3      CCC
3    K1      2.2    K1      AAA
4    K1      2.2    K2      BBB
5    K1      2.2    K3      CCC
6    K2      3.3    K1      AAA
7    K2      3.3    K2      BBB
8    K2      3.3    K3      CCC
```

Un exemple complet

Le tableur `chanson.xlsx` contient deux tables :

- `chanteurs` : nom, prénom, naissance et mort,
- `albums` : titre, année et chanteur.

La fonction `read_excel` (avec `openpyxl` ou `calamine`) permet de récupérer ces données dans des `DataFrame`.

```
1 df_chanteurs = pd.read_excel("data/chanson.xlsx", sheet_name="chant
2 df_albums = pd.read_excel("data/chanson.xlsx", sheet_name="albums")
```

```
1 print(df_chanteurs)
```

```
    prenom       nom  naissance    mort
0  Georges  Brassens       1921  1981.0
1     Léo     Ferré       1916  1993.0
2  Jacques      Brel       1929  1978.0
3   Renaud    Séchan       1952     NaN
```

```
1 print(df_albums.head(10)) # 10 premiers albums
```

```
                              titre  annee   prenom       nom
0             La Mauvaise Réputation   1952  Georges  Brassens
1                           Le Vent   1953  Georges  Brassens
2               Les Sabots d'Hélène   1954  Georges  Brassens
3       Je me suis fait tout petit   1956  Georges  Brassens
4                  Oncle Archibald   1957  Georges  Brassens
5                   Le Pornographe   1958  Georges  Brassens
6           Les Funérailles d'antan   1960  Georges  Brassens
7  Le temps ne fait rien à l'affaire   1961  Georges  Brassens
8     Les Trompettes de la renommée   1962  Georges  Brassens
9               Les Copains d'abord   1964  Georges  Brassens
```

Une jointure est possible sur le couple `(nom, prenom)` des chanteurs (ou simplement `nom` ici).

Voici une simple jointure à gauche :

```
print( # Clé primaire à 2 variables
    df_chanteurs.merge(df_albums, how="left", on=["prenom", "nom"])
)
```

```
     prenom       nom  naissance    mort                        titre   annee
0   Georges  Brassens       1921  1981.0      La Mauvaise Réputation  1952.0
1   Georges  Brassens       1921  1981.0                     Le Vent  1953.0
2   Georges  Brassens       1921  1981.0         Les Sabots d'Hélène  1954.0
3   Georges  Brassens       1921  1981.0  Je me suis fait tout petit  1956.0
4   Georges  Brassens       1921  1981.0              Oncle Archibald  1957.0
..      ...       ...        ...     ...                         ...     ...
60  Jacques      Brel       1929  1978.0                   J'arrive  1968.0
61  Jacques      Brel       1929  1978.0        L'Homme de la Mancha  1968.0
62  Jacques      Brel       1929  1978.0            Ne me quitte pas  1972.0
63  Jacques      Brel       1929  1978.0               Les Marquises  1977.0
64   Renaud    Séchan       1952     NaN                         NaN     NaN

[65 rows x 6 columns]
```

Le compte des albums sortis après 1968 dans `df_albums` par chanteur avec naissance et mort :

```
print(
    df_albums[df_albums.annee > 1968] # Filtre les lignes
    .groupby(["prenom", "nom"]) # Groupe par chanteur
    .titre.count() # Compte les albums par chanteur
    .reset_index() # Retour des indices en colonnes
    .rename(columns={"titre": "post68"}) # Renomme une colonne
    .merge(df_chanteurs, how="left", on=["prenom", "nom"]) # Left j
)
```

	prenom	nom	post68	naissance	mort
0	Georges	Brassens	3	1921.0	1981.0
1	Jacques	Brel	2	1929.0	1978.0
2	Juliette	Noureddine	12	NaN	NaN
3	Léo	Ferré	21	1916.0	1993.0

- Renaud n'a pas d'album après 1968 dans `df_albums`.
- Juliette a des albums après 1968 dans `df_albums` mais n'est pas présente dans `df_chanteurs`.

```
print( # Inner join
    df_albums[df_albums.annee > 1968]
    .groupby(["prenom", "nom"]).titre.count().reset_index()
    .rename(columns={"titre": "post68"})
    .merge(df_chanteurs, how="inner", on=["prenom", "nom"])
)
```

	prenom	nom	post68	naissance	mort
0	Georges	Brassens	3	1921	1981.0
1	Jacques	Brel	2	1929	1978.0
2	Léo	Ferré	21	1916	1993.0

```
print( # Outer join
    df_albums[df_albums.annee > 1968]
    .groupby(["prenom", "nom"]).titre.count().reset_index()
    .rename(columns={"titre": "post68"})
    .merge(df_chanteurs, how="outer", on=["prenom", "nom"])
)
```

	prenom	nom	post68	naissance	mort
0	Georges	Brassens	3.0	1921.0	1981.0
1	Jacques	Brel	2.0	1929.0	1978.0
2	Juliette	Noureddine	12.0	NaN	NaN
3	Léo	Ferré	21.0	1916.0	1993.0
4	Renaud	Séchan	NaN	1952.0	NaN

Données textuelles

De nombreux jeux de données contiennent des données textuelles. Ce type de données permet des opérations particulières (recherche, extraction, expressions régulières, …) que Pandas met à disposition grâce à l'attribut `str`.

```
s = pd.Series(["Bob", "Joy", "Ken"])
s.str.lower() # Passage en minuscules
```

```
0    bob
1    joy
2    ken
dtype: object
```

```
s.str.upper() # Passage en minuscules
```

```
0    BOB
1    JOY
2    KEN
dtype: object
```

Suppression d'espaces avec `strip`/`lstrip`/`rstrip`

```
1  s = pd.Series([" Bob!", "#Joy", "@Ken "])
2  s.str.strip().tolist() # Espaces de début et de fin
```

```
['Bob!', '#Joy', '@Ken']
```

```
1  s.str.lstrip().tolist() # Espaces de début
```

```
['Bob!', '#Joy', '@Ken ']
```

```
1  s.str.rstrip().tolist() # Espaces de fin
```

```
[' Bob!', '#Joy', '@Ken']
```

Suppression de caractères

```
1  s.str.strip(" !#@").tolist() # En début et en fin de chaîne
```

```
['Bob', 'Joy', 'Ken']
```

```
1  s.str.lstrip(" !#@").tolist() # En début de chaîne
```

```
['Bob!', 'Joy', 'Ken ']
```

Division de chaînes de caractères en listes

```
1  s = pd.Series(["Alpha_Blondy", "Bob_Marley", "Toots_Hibbert"])
2  s.str.split("_")
```

```
0      [Alpha, Blondy]
1        [Bob, Marley]
2    [Toots, Hibbert]
dtype: object
```

Division de chaînes de caractères en colonnes

```
1  print(
2      s.str.split("_", expand=True)
3  )
```

```
        0         1
0   Alpha    Blondy
1     Bob    Marley
2   Toots   Hibbert
```

Concaténation de chaînes de caractères

```
1  s = pd.Series(["Toots", "and", "the", "Maytals"])
2  s.str.cat() # Sans séparateur
```

```
'TootsandtheMaytals'
```

```
1  s.str.cat(sep=" ") # Avec séparateur
```

```
'Toots and the Maytals'
```

```
1  s.str.cat( # Avec d'autres chaînes de caractères
2      [f" - Mot {i}" for i in range(len(s))]
3  )
```

```
0      Toots - Mot 0
1        and - Mot 1
2        the - Mot 2
3    Maytals - Mot 3
dtype: object
```

Remplacement de caractères simples

```
1  s = pd.Series(["Alpha_Blondy", "Bob_Marley", "Toots_Hibbert"])
2  s.str.replace("_", " ")
```

```
0     Alpha Blondy
1       Bob Marley
2    Toots Hibbert
dtype: object
```

Remplacement par expression régulière

```
1  s.str.replace("^[A-Z][a-z]*_", "Rasta ", regex=True)
```

```
0     Rasta Blondy
1     Rasta Marley
2    Rasta Hibbert
dtype: object
```

Expressions régulières

Les expressions régulières (*regex*) obéissent à une syntaxe bien définie et permettent de décrire un ensemble de chaînes de caractères possibles.

La syntaxe peut vite devenir compliquée mais il s'agit d'un outil très puissant pour la manipulation de chaînes de caractères.

```
(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.
[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))
```

Quelques éléments d'expressions régulières :

- `^Pyt` Commence par "Pyt"
- `hon$` Termine par "hon"
- `.` Tout caractère sauf la fin de ligne
- `[Abc]` Caractère "A", "b" ou "c"
- `[^Abc]` Caractère différent de "A", "b" et "c"
- `[A-Z]`/`[a-z]` Lettre majuscule/minuscule
- `[0-9]` Chiffre
- `\w` Caractère alphanumérique (`\W` non alphanumérique)
- `\s` Caractère blanc (espace, tabulation, …) (`\S` non blanc)

- `P?` Caractère “P” 0 ou 1 fois
- `P+` Caractère “P” 1 fois ou plus
- `P*` Caractère “P” 0 fois ou plus
- `(A[a-z])` Les parenthèses délimitent un groupe
- `\(` et `\)` Caractères “(” et “)”
- `\w+(?=0)` Alphanumériques suivis de “0”
- `\w+(?!0)` Alphanumériques non suivis de “0”
- `(?<=0)\w+` Alphanumériques précédés par “0”
- `(?<!0)\w+` Alphanumériques non précédés par “0”
- …

Contenu d’une chaîne de caractères

```
1  s = pd.Series(["1", "2", "3a", "3b", "03c", "4dx"])
2  s.str.contains("[0-9][a-z]").tolist()
```

```
[False, False, True, True, True, True]
```

Correspondance du début d’une chaîne de caractères

```
1  s.str.match("[0-9][a-z]").tolist()
```

```
[False, False, True, True, False, True]
```

Correspondance exacte d’une chaîne de caractères

```
1  s.str.fullmatch("[0-9][a-z]").tolist()
```

```
[False, False, True, True, False, False]
```

Les groupes définis par des parenthèses dans une expression régulière peuvent être extraits par la méthode `extract`.

```
s = pd.Series([
    "Score de Bob: 31415",
    "Score de Joy: 42024",
])

print(
    s.str.extract("^Score de (\w+): ([0-9]+)")
)
```

```
     0      1
0  Bob  31415
1  Joy  42024
```

À vous de jouer !