eBook Gratuit

APPRENEZ pandas

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#pandas

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: pandas

It is an unofficial and free pandas ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official pandas.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec les pandas

Remarques

Pandas est un package Python fournissant des structures de données rapides, flexibles et expressives conçues pour rendre le travail avec des données «relationnelles» ou «étiquetées» à la fois simples et intuitives. Il se veut être le composant fondamental de haut niveau pour effectuer des analyses de données pratiques dans le monde réel en Python.

La documentation officielle Pandas peut être trouvée ici .

Versions

Les pandas

Version	Date de sortie
0.19.1	2016-11-03
0.19.0	2016-10-02
0.18.1	2016-05-03
0.18.0	2016-03-13
0,17,1	2015-11-21
0,17,0	2015-10-09
0,16,2	2015-06-12
0.16.1	2015-05-11
0,16,0	2015-03-22
0.15.2	2014-12-12
0.15.1	2014-11-09
0,15,0	2014-10-18
0.14.1	2014-07-11
0,14,0	2014-05-31
0.13.1	2014-02-03
0.13.0	2014-01-03

Version	Date de sortie
0.12.0	2013-07-23

Examples

Installation ou configuration

Des instructions détaillées sur la mise en place ou l'installation de pandas sont disponibles dans la documentation officielle .

Installer des pandas avec Anaconda

Installer des pandas et le reste de la pile NumPy et SciPy peut être un peu difficile pour les utilisateurs inexpérimentés.

La manière la plus simple d'installer non seulement les pandas, mais aussi Python et les paquets les plus populaires constituant la pile SciPy (IPython, NumPy, Matplotlib, ...) est avec Anaconda , une plateforme multi-plateforme (Linux, Mac OS X, Windows) Distribution Python pour l'analyse de données et le calcul scientifique.

Après avoir exécuté un programme d'installation simple, l'utilisateur aura accès aux pandas et au reste de la pile SciPy sans avoir besoin d'installer autre chose, et sans avoir à attendre que des logiciels soient compilés.

Vous trouverez les instructions d'installation d'Anaconda ici .

Une liste complète des paquets disponibles dans le cadre de la distribution d'Anaconda peut être trouvée ici .

Un avantage supplémentaire de l'installation avec Anaconda est que vous n'avez pas besoin des droits d'administrateur pour l'installer, il sera installé dans le répertoire personnel de l'utilisateur, et cela rend également inutile de supprimer Anaconda ultérieurement (il suffit de supprimer ce dossier).

Installer des pandas avec Miniconda

La section précédente décrivait comment installer les pandas dans le cadre de la distribution Anaconda. Cependant, cette approche signifie que vous allez installer plus de cent paquets et que vous devrez télécharger le programme d'installation de quelques centaines de mégaoctets.

Si vous voulez avoir plus de contrôle sur les paquets, ou si vous avez une bande passante Internet limitée, installer des pandas avec Miniconda peut être une meilleure solution.

Conda est le gestionnaire de paquetages sur lequel la distribution Anaconda est construite. C'est un gestionnaire de paquets à la fois multi-plateforme et indépendant du langage (il peut jouer un rôle similaire à une combinaison pip et virtualenv).

Miniconda vous permet de créer une installation Python autonome minimale, puis d'utiliser la

commande Conda pour installer des packages supplémentaires.

Tout d'abord, vous aurez besoin de Conda pour être installé et télécharger et exécuter le Miniconda le fera pour vous. L'installateur peut être trouvé ici .

L'étape suivante consiste à créer un nouvel environnement de conda (ceux-ci sont analogues à virtualenv, mais ils vous permettent également de spécifier précisément la version de Python à installer). Exécutez les commandes suivantes à partir d'une fenêtre de terminal:

```
conda create -n name_of_my_env python
```

Cela créera un environnement minimal avec seulement Python installé. Pour vous mettre dans cet environnement, exécutez:

```
source activate name_of_my_env
```

Sous Windows, la commande est la suivante:

```
activate name_of_my_env
```

La dernière étape requise consiste à installer des pandas. Cela peut être fait avec la commande suivante:

```
conda install pandas
```

Pour installer une version de pandas spécifique:

```
conda install pandas=0.13.1
```

Pour installer d'autres packages, IPython par exemple:

```
conda install ipython
```

Pour installer la distribution complète d'Anaconda:

```
conda install anaconda
```

Si vous avez besoin de paquets disponibles pour pip mais pas conda, installez simplement pip et utilisez pip pour installer ces paquets:

```
conda install pip
pip install django
```

Généralement, vous installez des pandas avec l'un des gestionnaires de paquets.

exemple pip:

```
pip install pandas
```

Cela nécessitera probablement l'installation d'un certain nombre de dépendances, y compris NumPy, nécessitera un compilateur pour compiler les bits de code requis, et cela peut prendre quelques minutes.

Installer via anaconda

Commencez par télécharger anaconda du site Continuum. Soit via l'installateur graphique (Windows / OSX) ou en exécutant un script shell (OSX / Linux). Cela comprend les pandas!

Si vous ne voulez pas que les 150 paquets soient regroupés dans anaconda, vous pouvez installer miniconda . Soit via l'installateur graphique (Windows) ou le script shell (OSX / Linux).

Installez les pandas sur miniconda en utilisant:

```
conda install pandas
```

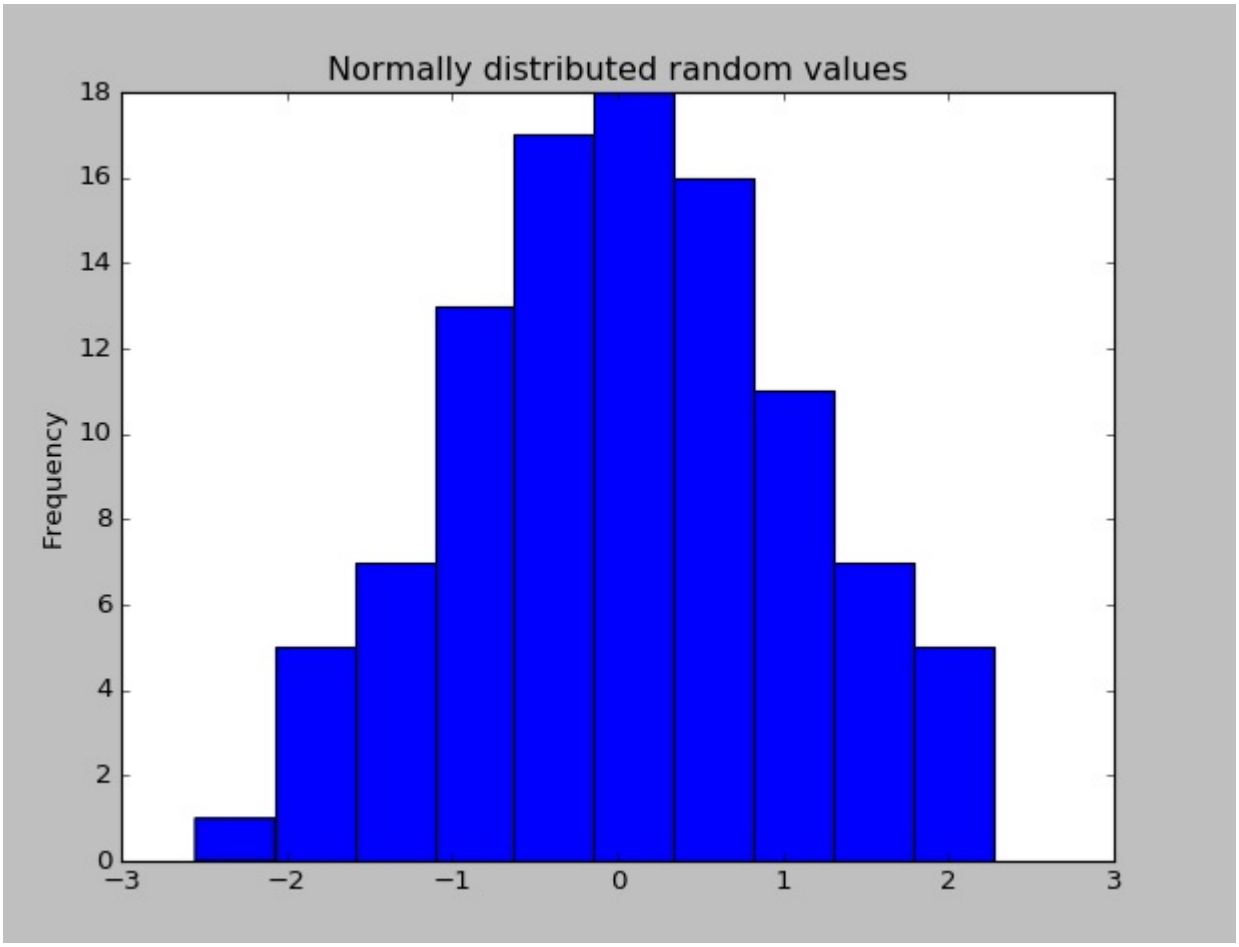
Pour mettre à jour les pandas à la dernière version en utilisation anaconda ou miniconda:

```
conda update pandas
```

Bonjour le monde

Une fois Pandas installé, vous pouvez vérifier s'il fonctionne correctement en créant un ensemble de données réparties de manière aléatoire et en traçant son histogramme.

```
import pandas as pd  # This is always assumed but is included here as an introduction.
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)

values = np.random.randn(100) # array of normally distributed random numbers
s = pd.Series(values) # generate a pandas series
s.plot(kind='hist', title='Normally distributed random values') # hist computes distribution
plt.show()
```



Vérifiez certaines des statistiques des données (moyenne, écart-type, etc.)

```
s.describe()
# Output: count    100.000000
# mean       0.059808
# std        1.012960
# min       -2.552990
# 25%       -0.643857
# 50%        0.094096
# 75%        0.737077
# max        2.269755
# dtype: float64
```

Statistiques descriptives

Les statistiques descriptives (moyenne, écart-type, nombre d'observations, minimum, maximum et quartiles) des colonnes numériques peuvent être calculées à l'aide de la méthode `.describe()` , qui renvoie un ensemble de données descriptives de pandas.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 1, 4, 3, 5, 2, 3, 4, 1],
                           'B': [12, 14, 11, 16, 18, 18, 22, 13, 21, 17],
                           'C': ['a', 'a', 'b', 'a', 'b', 'c', 'b', 'a', 'b', 'a']})

In [2]: df
Out[2]:
   A   B  C
0  1  12  a
```

```
1  2  14  a
2  1  11  b
3  4  16  a
4  3  18  b
5  5  18  c
6  2  22  b
7  3  13  a
8  4  21  b
9  1  17  a

In [3]: df.describe()
Out[3]:
                A          B
count  10.000000  10.000000
mean    2.600000  16.200000
std     1.429841   3.705851
min     1.000000  11.000000
25%     1.250000  13.250000
50%     2.500000  16.500000
75%     3.750000  18.000000
max     5.000000  22.000000
```

Notez que puisque `C` n'est pas une colonne numérique, il est exclu de la sortie.

```
In [4]: df['C'].describe()
Out[4]:
count     10
unique     3
freq       5
Name: C, dtype: object
```

Dans ce cas, la méthode résume les données catégorielles par le nombre d'observations, le nombre d'éléments uniques, le mode et la fréquence du mode.

Lire Commencer avec les pandas en ligne: https://riptutorial.com/fr/pandas/topic/796/commencer-avec-les-pandas

Chapitre 2: Ajout à DataFrame

Examples

Ajout d'une nouvelle ligne à DataFrame

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

In [2]: df = pd.DataFrame(columns = ['A', 'B', 'C'])

In [3]: df
Out[3]:
Empty DataFrame
Columns: [A, B, C]
Index: []
```

Ajout d'une ligne par une valeur de colonne unique:

```
In [4]: df.loc[0, 'A'] = 1

In [5]: df
Out[5]:
   A    B    C
0  1  NaN  NaN
```

Ajout d'une ligne, liste de valeurs donnée:

```
In [6]: df.loc[1] = [2, 3, 4]

In [7]: df
Out[7]:
   A    B    C
0  1  NaN  NaN
1  2    3    4
```

Ajouter une ligne à un dictionnaire:

```
In [8]: df.loc[2] = {'A': 3, 'C': 9, 'B': 9}

In [9]: df
Out[9]:
   A    B    C
0  1  NaN  NaN
1  2    3    4
2  3    9    9
```

La première entrée de .loc [] est l'index. Si vous utilisez un index existant, vous écraserez les valeurs de cette ligne:

```
In [17]: df.loc[1] = [5, 6, 7]
```

```
In [18]: df
Out[18]:
   A    B    C
0  1  NaN  NaN
1  5    6    7
2  3    9    9


In [19]: df.loc[0, 'B'] = 8

In [20]: df
Out[20]:
   A  B    C
0  1  8  NaN
1  5  6    7
2  3  9    9
```

Ajouter un DataFrame à un autre DataFrame

Supposons que nous ayons les deux DataFrames suivants:

```
In [7]: df1
Out[7]:
    A   B
0  a1  b1
1  a2  b2

In [8]: df2
Out[8]:
    B   C
0  b1  c1
```

Les deux DataFrames ne doivent pas nécessairement avoir le même ensemble de colonnes. La méthode append ne modifie aucun des DataFrames d'origine. Au lieu de cela, il renvoie un nouveau DataFrame en ajoutant les deux originaux. Ajouter un DataFrame à un autre est assez simple:

```
In [9]: df1.append(df2)
Out[9]:
     A   B    C
0   a1  b1  NaN
1   a2  b2  NaN
0  NaN  b1   c1
```

Comme vous pouvez le voir, il est possible d'avoir des index en double (0 dans cet exemple). Pour éviter ce problème, vous pouvez demander à Pandas de réindexer le nouveau DataFrame pour vous:

```
In [10]: df1.append(df2, ignore_index = True)
Out[10]:
     A   B    C
0   a1  b1  NaN
1   a2  b2  NaN
2  NaN  b1   c1
```

Lire Ajout à DataFrame en ligne: https://riptutorial.com/fr/pandas/topic/6456/ajout-a-dataframe

Chapitre 3: Analyse: tout rassembler et prendre des décisions

Examples

Analyse de quintile: avec des données aléatoires

L'analyse de quintile est un cadre commun pour évaluer l'efficacité des facteurs de sécurité.

Qu'est-ce qu'un facteur

Un facteur est une méthode de notation / classement des ensembles de titres. Pour un moment donné et pour un ensemble particulier de titres, un facteur peut être représenté comme une série de pandas dans laquelle l'index est un tableau des identificateurs de sécurité et les valeurs sont les scores ou les rangs.

Si nous prenons des scores factoriels au fil du temps, nous pouvons, à chaque instant, diviser l'ensemble des titres en 5 compartiments ou quintiles égaux, en fonction de l'ordre des scores factoriels. Il n'y a rien de particulièrement sacré dans le nombre 5. Nous aurions pu utiliser 3 ou 10. Mais nous utilisons 5 souvent. Enfin, nous suivons la performance de chacun des cinq compartiments pour déterminer s'il ya une différence significative dans les rendements. Nous avons tendance à nous concentrer davantage sur la différence de rendement du seau avec le rang le plus élevé par rapport à celui du rang le plus bas.

Commençons par définir certains paramètres et générer des données aléatoires.

Pour faciliter l'expérimentation avec la mécanique, nous fournissons un code simple pour créer des données aléatoires afin de nous donner une idée de son fonctionnement.

Les données aléatoires incluent

- **Retours** : générer des retours aléatoires pour un nombre spécifié de titres et de périodes.
- **Signaux** : génèrent des signaux aléatoires pour un nombre spécifié de titres et de périodes et avec le niveau de corrélation prescrit avec les **retours** . Pour qu'un facteur soit utile, il doit y avoir des informations ou une corrélation entre les scores / rangs et les retours ultérieurs. S'il n'y avait pas de corrélation, nous le verrions. Ce serait un bon exercice pour le lecteur, dupliquer cette analyse avec des données aléatoires générées avec `0` corrélation.

Initialisation

```
import pandas as pd
import numpy as np
```

```
num_securities = 1000
num_periods = 1000
period_frequency = 'W'
start_date = '2000-12-31'

np.random.seed([3,1415])

means = [0, 0]
covariance = [[  1., 5e-3],
              [5e-3,   1.]]

# generates to sets of data m[0] and m[1] with ~0.005 correlation
m = np.random.multivariate_normal(means, covariance,
                                  (num_periods, num_securities)).T
```

Générons maintenant un index de séries temporelles et un index représentant les identifiants de sécurité. Ensuite, utilisez-les pour créer des diagrammes de données pour les retours et les signaux

```
ids = pd.Index(['s{:05d}'.format(s) for s in range(num_securities)], 'ID')
tidx = pd.date_range(start=start_date, periods=num_periods, freq=period_frequency)
```

Je divise `m[0]` par `25` pour réduire à quelque chose qui ressemble à des rendements boursiers. J'ajoute également `1e-7` pour donner un rendement moyen positif modeste.

```
security_returns = pd.DataFrame(m[0] / 25 + 1e-7, tidx, ids)
security_signals = pd.DataFrame(m[1], tidx, ids)
```

`pd.qcut` - Create Quintile Buckets

Utilisons `pd.qcut` pour diviser mes signaux en `pd.qcut` quintiles pour chaque période.

```
def qcut(s, q=5):
    labels = ['q{}'.format(i) for i in range(1, 6)]
    return pd.qcut(s, q, labels=labels)

cut = security_signals.stack().groupby(level=0).apply(qcut)
```

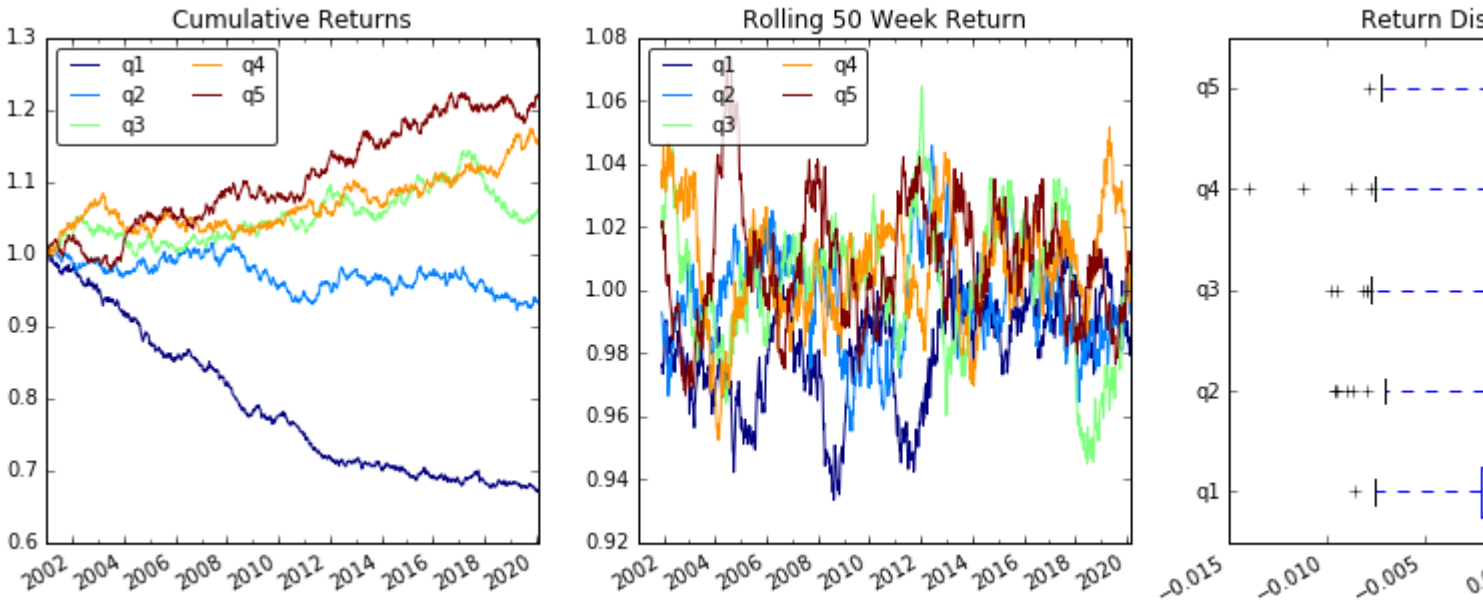
Utilisez ces coupes comme index sur nos retours

```
returns_cut = security_returns.stack().rename('returns') \
    .to_frame().set_index(cut, append=True) \
    .swaplevel(2, 1).sort_index().squeeze() \
    .groupby(level=[0, 1]).mean().unstack()
```

Une analyse

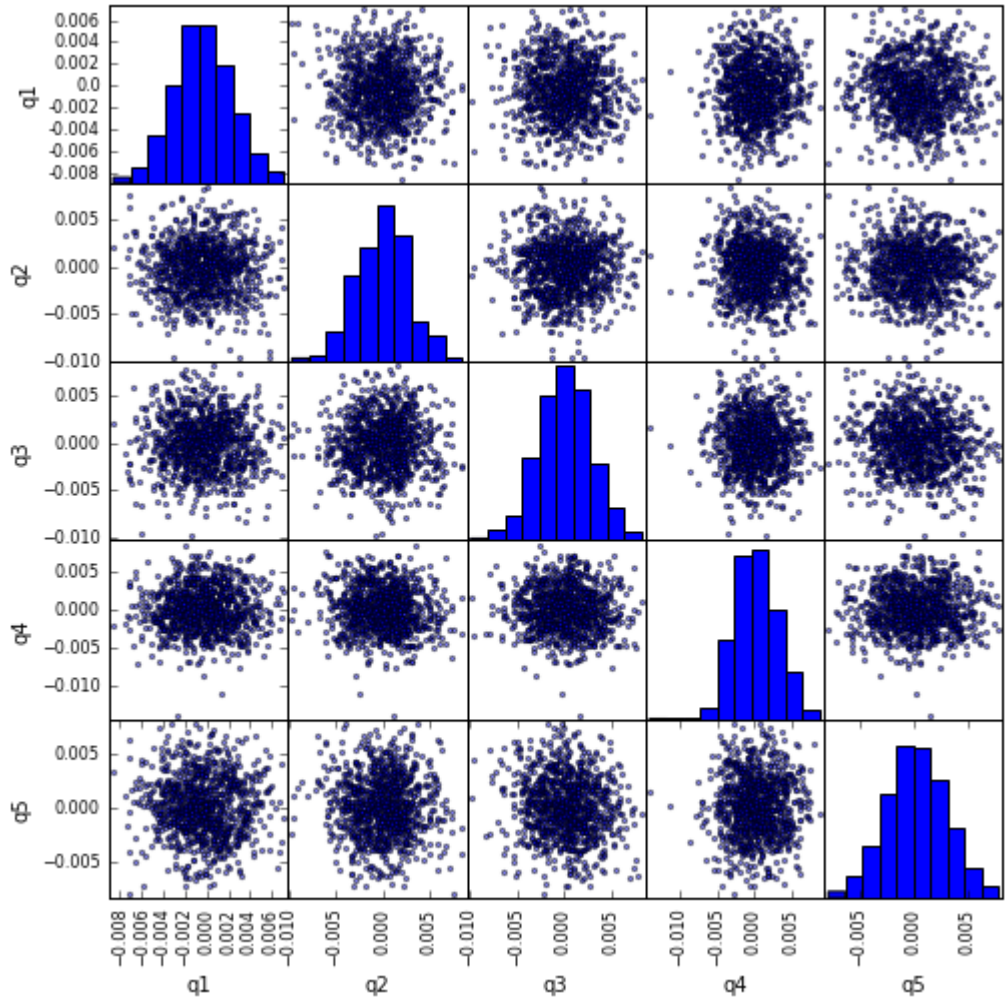
Retours de parcelles

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(15, 5))
ax1 = plt.subplot2grid((1,3), (0,0))
ax2 = plt.subplot2grid((1,3), (0,1))
ax3 = plt.subplot2grid((1,3), (0,2))

# Cumulative Returns
returns_cut.add(1).cumprod() \
    .plot(colormap='jet', ax=ax1, title="Cumulative Returns")
leg1 = ax1.legend(loc='upper left', ncol=2, prop={'size': 10}, fancybox=True)
leg1.get_frame().set_alpha(.8)

# Rolling 50 Week Return
returns_cut.add(1).rolling(50).apply(lambda x: x.prod()) \
    .plot(colormap='jet', ax=ax2, title="Rolling 50 Week Return")
leg2 = ax2.legend(loc='upper left', ncol=2, prop={'size': 10}, fancybox=True)
leg2.get_frame().set_alpha(.8)

# Return Distribution
returns_cut.plot.box(vert=False, ax=ax3, title="Return Distribution")

fig.autofmt_xdate()

plt.show()
```



Visualiser la corrélation de quintile avec `scatter_matrix`

```
from pandas.tools.plotting import scatter_matrix

scatter_matrix(returns_cut, alpha=0.5, figsize=(8, 8), diagonal='hist')
plt.show()
```



Calculer et visualiser Maximum Draw Down

```
def max_dd(returns):
    """returns is a series"""
    r = returns.add(1).cumprod()
    dd = r.div(r.cummax()).sub(1)
    mdd = dd.min()
    end = dd.argmin()
    start = r.loc[:end].argmax()
    return mdd, start, end

def max_dd_df(returns):
    """returns is a dataframe"""
    series = lambda x: pd.Series(x, ['Draw Down', 'Start', 'End'])
    return returns.apply(max_dd).apply(series)
```

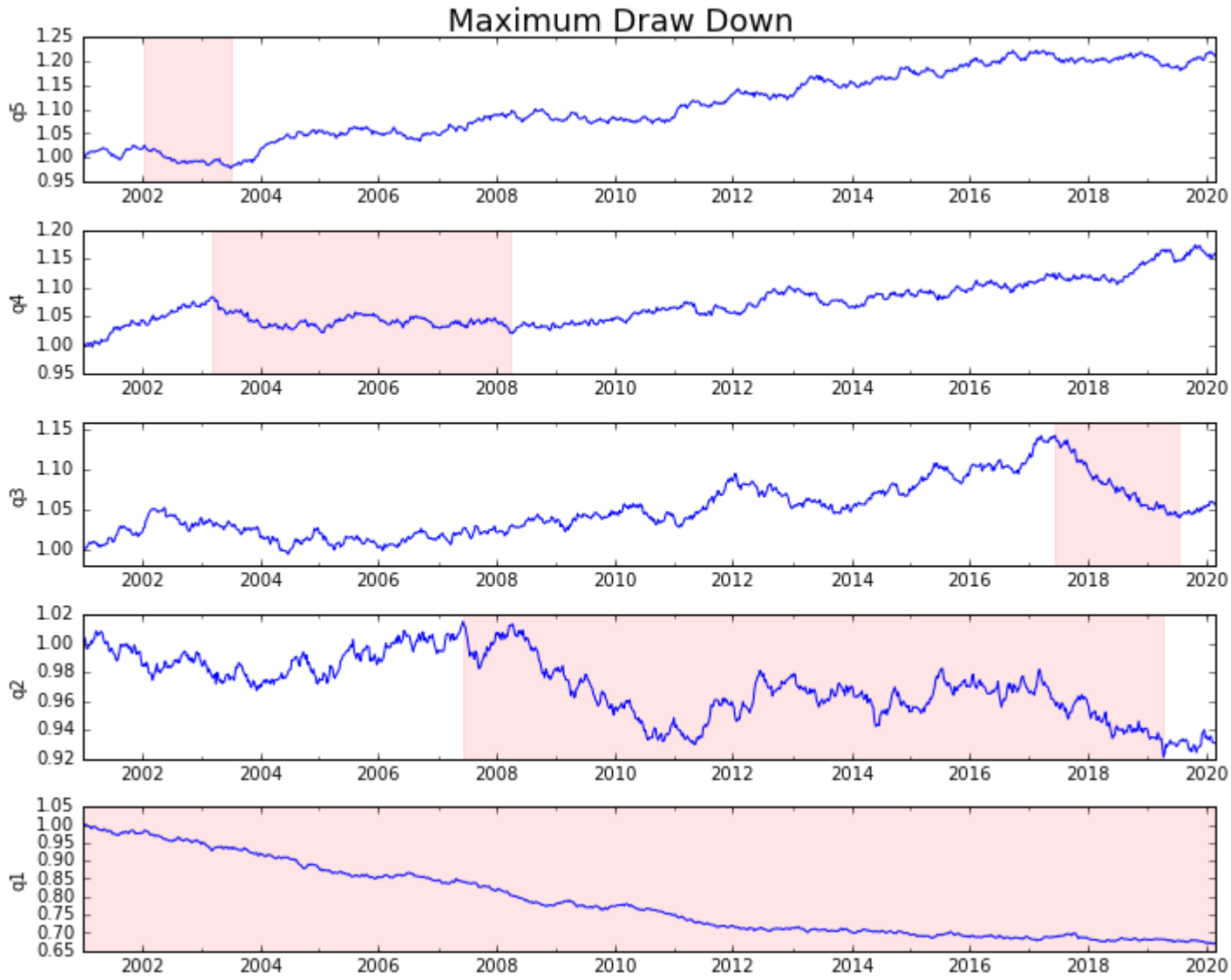
A quoi ça ressemble

```
max_dd_df(returns_cut)
```

	Draw Down	Start	End
q1	-0.333527	2001-01-07	2020-02-16
q2	-0.092659	2007-06-10	2019-04-14
q3	-0.089682	2017-06-11	2019-07-21
q4	-0.058225	2003-03-16	2008-03-30
q5	-0.046822	2002-01-20	2003-07-06

Plaquons-le

```
draw_downs = max_dd_df(returns_cut)

fig, axes = plt.subplots(5, 1, figsize=(10, 8))
for i, ax in enumerate(axes[::-1]):
    returns_cut.iloc[:, i].add(1).cumprod().plot(ax=ax)
    sd, ed = draw_downs[['Start', 'End']].iloc[i]
    ax.axvspan(sd, ed, alpha=0.1, color='r')
    ax.set_ylabel(returns_cut.columns[i])

fig.suptitle('Maximum Draw Down', fontsize=18)
fig.tight_layout()
plt.subplots_adjust(top=.95)
```



Calculer des statistiques

Il existe de nombreuses statistiques potentielles que nous pouvons inclure. En voici quelques-uns, mais montrez comment nous pouvons simplement intégrer de nouvelles statistiques dans notre résumé.

```
def frequency_of_time_series(df):
    start, end = df.index.min(), df.index.max()
    delta = end - start
    return round((len(df) - 1.) * 365.25 / delta.days, 2)

def annualized_return(df):
    freq = frequency_of_time_series(df)
    return df.add(1).prod() ** (1 / freq) - 1

def annualized_volatility(df):
    freq = frequency_of_time_series(df)
    return df.std().mul(freq ** .5)

def sharpe_ratio(df):
    return annualized_return(df) / annualized_volatility(df)
```

```
def describe(df):
    r = annualized_return(df).rename('Return')
    v = annualized_volatility(df).rename('Volatility')
    s = sharpe_ratio(df).rename('Sharpe')
    skew = df.skew().rename('Skew')
    kurt = df.kurt().rename('Kurtosis')
    desc = df.describe().T

    return pd.concat([r, v, s, skew, kurt, desc], axis=1).T.drop('count')
```

Nous finirons par utiliser uniquement la fonction de `describe` car elle rassemble tous les autres.

```
describe(returns_cut)
```

	q1	q2	q3	q4	q5
Return	-0.007609	-0.001375	0.001067	0.002821	0.003687
Volatility	0.019584	0.020445	0.020629	0.021185	0.020172
Sharpe	-0.388525	-0.067278	0.051709	0.133176	0.182792
Skew	0.040430	-0.085828	-0.078071	-0.067522	0.005652
Kurtosis	-0.174206	0.203038	0.026385	0.370249	-0.160678
mean	-0.000395	-0.000068	0.000060	0.000151	0.000196
std	0.002711	0.002830	0.002856	0.002933	0.002792
min	-0.008608	-0.009614	-0.009845	-0.014037	-0.007913
25%	-0.002196	-0.002018	-0.001956	-0.001833	-0.001694
50%	-0.000434	0.000065	0.000210	0.000029	0.000146
75%	0.001444	0.001768	0.001989	0.002107	0.002081
max	0.007070	0.008432	0.008100	0.008687	0.007791

Ceci n'est pas censé être complet. Il est destiné à rassembler de nombreuses fonctionnalités des pandas et à démontrer comment vous pouvez les utiliser pour répondre à des questions importantes pour vous. C'est un sous-ensemble des types de paramètres que j'utilise pour évaluer l'efficacité des facteurs quantitatifs.

Lire Analyse: tout rassembler et prendre des décisions en ligne:
https://riptutorial.com/fr/pandas/topic/5238/analyse--tout-rassembler-et-prendre-des-decisions

Chapitre 4: Calendriers de vacances

Examples

Créer un calendrier personnalisé

Voici comment créer un calendrier personnalisé. L'exemple donné est un calendrier français - il fournit donc de nombreux exemples.

```
from pandas.tseries.holiday import AbstractHolidayCalendar, Holiday, EasterMonday, Easter
from pandas.tseries.offsets import Day, CustomBusinessDay

class FrBusinessCalendar(AbstractHolidayCalendar):
    """ Custom Holiday calendar for France based on
        https://en.wikipedia.org/wiki/Public_holidays_in_France
      - 1 January: New Year's Day
      - Moveable: Easter Monday (Monday after Easter Sunday)
      - 1 May: Labour Day
      - 8 May: Victory in Europe Day
      - Moveable Ascension Day (Thursday, 39 days after Easter Sunday)
      - 14 July: Bastille Day
      - 15 August: Assumption of Mary to Heaven
      - 1 November: All Saints' Day
      - 11 November: Armistice Day
      - 25 December: Christmas Day
    """
    rules = [
        Holiday('New Years Day', month=1, day=1),
        EasterMonday,
        Holiday('Labour Day', month=5, day=1),
        Holiday('Victory in Europe Day', month=5, day=8),
        Holiday('Ascension Day', month=1, day=1, offset=[Easter(), Day(39)]),
        Holiday('Bastille Day', month=7, day=14),
        Holiday('Assumption of Mary to Heaven', month=8, day=15),
        Holiday('All Saints Day', month=11, day=1),
        Holiday('Armistice Day', month=11, day=11),
        Holiday('Christmas Day', month=12, day=25)
    ]
```

Utiliser un calendrier personnalisé

Voici comment utiliser le calendrier personnalisé.

Obtenez les vacances entre deux dates

```
import pandas as pd
from datetime import date

# Creating some boundaries
year = 2016
start = date(year, 1, 1)
```

```
end = start + pd.offsets.MonthEnd(12)

# Creating a custom calendar
cal = FrBusinessCalendar()
# Getting the holidays (off-days) between two dates
cal.holidays(start=start, end=end)

# DatetimeIndex(['2016-01-01', '2016-03-28', '2016-05-01', '2016-05-05',
#                '2016-05-08', '2016-07-14', '2016-08-15', '2016-11-01',
#                '2016-11-11', '2016-12-25'],
#               dtype='datetime64[ns]', freq=None)
```

Compter le nombre de jours ouvrables entre deux dates

Il est parfois utile d'obtenir le nombre de jours de travail par mois, quelle que soit l'année ou le passé. Voici comment procéder avec un calendrier personnalisé.

```
from pandas.tseries.offsets import CDay

# Creating a series of dates between the boundaries
# by using the custom calendar
se = pd.bdate_range(start=start,
                    end=end,
                    freq=CDay(calendar=cal)).to_series()
# Counting the number of working days by month
se.groupby(se.dt.month).count().head()

# 1    20
# 2    21
# 3    22
# 4    21
# 5    21
```

Lire Calendriers de vacances en ligne: https://riptutorial.com/fr/pandas/topic/7976/calendriers-de-vacances

Chapitre 5: Création de DataFrames

Introduction

DataFrame est une structure de données fournie par la bibliothèque pandas, à l'exception de *Series* & *Panel* . C'est une structure à deux dimensions et peut être comparée à une table de lignes et de colonnes.

Chaque ligne peut être identifiée par un index entier (0..N) ou une étiquette explicitement définie lors de la création d'un objet DataFrame. Chaque colonne peut être de type distinct et identifiée par une étiquette.

Cette rubrique couvre différentes façons de créer / créer un objet DataFrame. Ex. des tableaux Numpy, de la liste des tuples, du dictionnaire.

Examples

Créer un exemple de DataFrame

```
import pandas as pd
```

Créez un DataFrame à partir d'un dictionnaire, contenant deux colonnes: des `numbers` et des `colors` . Chaque clé représente un nom de colonne et la valeur est une série de données, le contenu de la colonne:

```
df = pd.DataFrame({'numbers': [1, 2, 3], 'colors': ['red', 'white', 'blue']})
```

Afficher le contenu du dataframe:

```
print(df)
# Output:
#   colors  numbers
# 0    red        1
# 1  white        2
# 2   blue        3
```

Commandes de Pandas colonnes par ordre alphabétique comme `dict` ne sont pas ordonnés. Pour spécifier la commande, utilisez le paramètre `columns` .

```
df = pd.DataFrame({'numbers': [1, 2, 3], 'colors': ['red', 'white', 'blue']},
                  columns=['numbers', 'colors'])

print(df)
# Output:
#    numbers colors
# 0        1    red
# 1        2  white
```

```
# 2        3   blue
```

Créer un exemple de DataFrame en utilisant Numpy

Créez un `DataFrame` de nombres aléatoires:

```
import numpy as np
import pandas as pd

# Set the seed for a reproducible sample
np.random.seed(0)

df = pd.DataFrame(np.random.randn(5, 3), columns=list('ABC'))

print(df)
# Output:
#           A         B         C
# 0  1.764052  0.400157  0.978738
# 1  2.240893  1.867558 -0.977278
# 2  0.950088 -0.151357 -0.103219
# 3  0.410599  0.144044  1.454274
# 4  0.761038  0.121675  0.443863
```

Créez un `DataFrame` avec des entiers:

```
df = pd.DataFrame(np.arange(15).reshape(5,3),columns=list('ABC'))

print(df)
# Output:
#     A   B   C
# 0   0   1   2
# 1   3   4   5
# 2   6   7   8
# 3   9  10  11
# 4  12  13  14
```

Créez un `DataFrame` et incluez les nans (`NaT, NaN, 'nan', None`) entre les colonnes et les lignes:

```
df = pd.DataFrame(np.arange(48).reshape(8,6),columns=list('ABCDEF'))

print(df)
# Output:
#     A   B   C   D   E   F
# 0   0   1   2   3   4   5
# 1   6   7   8   9  10  11
# 2  12  13  14  15  16  17
# 3  18  19  20  21  22  23
# 4  24  25  26  27  28  29
# 5  30  31  32  33  34  35
# 6  36  37  38  39  40  41
# 7  42  43  44  45  46  47

df.ix[::2,0] = np.nan # in column 0, set elements with indices 0,2,4, ... to NaN
df.ix[::4,1] = pd.NaT # in column 1, set elements with indices 0,4, ... to np.NaT
df.ix[:3,2] = 'nan'   # in column 2, set elements with index from 0 to 3 to 'nan'
df.ix[:,5] = None     # in column 5, set all elements to None
```

```
df.ix[5,:] = None     # in row 5, set all elements to None
df.ix[7,:] = np.nan   # in row 7, set all elements to NaN

print(df)
# Output:
#     A     B     C   D   E     F
# 0 NaN   NaT   nan   3   4  None
# 1   6     7   nan   9  10  None
# 2 NaN    13   nan  15  16  None
# 3  18    19   nan  21  22  None
# 4 NaN   NaT    26  27  28  None
# 5 NaN  None  None NaN NaN  None
# 6 NaN    37    38  39  40  None
# 7 NaN   NaN   NaN NaN NaN   NaN
```

Créer un exemple de DataFrame à partir de plusieurs collections à l'aide d'un dictionnaire

```
import pandas as pd
import numpy as np

np.random.seed(123)
x = np.random.standard_normal(4)
y = range(4)
df = pd.DataFrame({'X':x, 'Y':y})
>>> df
          X  Y
0 -1.085631  0
1  0.997345  1
2  0.282978  2
3 -1.506295  3
```

Créer un DataFrame à partir d'une liste de tuples

Vous pouvez créer un DataFrame à partir d'une liste de tuples simples, et même choisir les éléments spécifiques des tuples à utiliser. Ici, nous allons créer un DataFrame en utilisant toutes les données de chaque tuple, à l'exception du dernier élément.

```
import pandas as pd

data = [
('p1', 't1', 1, 2),
('p1', 't2', 3, 4),
('p2', 't1', 5, 6),
('p2', 't2', 7, 8),
('p2', 't3', 2, 8)
]

df = pd.DataFrame(data)

print(df)
#     0   1  2  3
# 0  p1  t1  1  2
# 1  p1  t2  3  4
# 2  p2  t1  5  6
# 3  p2  t2  7  8
```

```
# 4  p2  t3  2  8
```

Créer un DataFrame à partir d'un dictionnaire de listes

Créez un DataFrame à partir de plusieurs listes en transmettant un dict dont les listes de valeurs. Les clés du dictionnaire sont utilisées comme étiquettes de colonne. Les listes peuvent aussi être des ndarrays. Les listes / ndarrays doivent tous avoir la même longueur.

```
import pandas as pd

# Create DF from dict of lists/ndarrays
df = pd.DataFrame({'A' : [1, 2, 3, 4],
                   'B' : [4, 3, 2, 1]})
df
# Output:
#       A  B
#    0  1  4
#    1  2  3
#    2  3  2
#    3  4  1
```

Si les tableaux n'ont pas la même longueur, une erreur est générée

```
df = pd.DataFrame({'A' : [1, 2, 3, 4], 'B' : [5, 5, 5]}) # a ValueError is raised
```

Utiliser ndarrays

```
import pandas as pd
import numpy as np

np.random.seed(123)
x = np.random.standard_normal(4)
y = range(4)
df = pd.DataFrame({'X':x, 'Y':y})
df
# Output:           X  Y
#         0 -1.085631  0
#         1  0.997345  1
#         2  0.282978  2
#         3 -1.506295  3
```

Voir les détails supplémentaires sur: http://pandas.pydata.org/pandas-docs/stable/dsintro.html#from-dict-of-ndarrays-lists

Créer un exemple de DataFrame avec datetime

```
import pandas as pd
import numpy as np

np.random.seed(0)
# create an array of 5 dates starting at '2015-02-24', one per minute
rng = pd.date_range('2015-02-24', periods=5, freq='T')
df = pd.DataFrame({ 'Date': rng, 'Val': np.random.randn(len(rng)) })
```

```
print (df)
# Output:
#                   Date       Val
# 0 2015-02-24 00:00:00  1.764052
# 1 2015-02-24 00:01:00  0.400157
# 2 2015-02-24 00:02:00  0.978738
# 3 2015-02-24 00:03:00  2.240893
# 4 2015-02-24 00:04:00  1.867558

# create an array of 5 dates starting at '2015-02-24', one per day
rng = pd.date_range('2015-02-24', periods=5, freq='D')
df = pd.DataFrame({ 'Date': rng, 'Val' : np.random.randn(len(rng))})

print (df)
# Output:
#         Date       Val
# 0 2015-02-24 -0.977278
# 1 2015-02-25  0.950088
# 2 2015-02-26 -0.151357
# 3 2015-02-27 -0.103219
# 4 2015-02-28  0.410599

# create an array of 5 dates starting at '2015-02-24', one every 3 years
rng = pd.date_range('2015-02-24', periods=5, freq='3A')
df = pd.DataFrame({ 'Date': rng, 'Val' : np.random.randn(len(rng))})

print (df)
# Output:
#         Date       Val
# 0 2015-12-31  0.144044
# 1 2018-12-31  1.454274
# 2 2021-12-31  0.761038
# 3 2024-12-31  0.121675
# 4 2027-12-31  0.443863
```

DataFrame avec `DatetimeIndex` :

```
import pandas as pd
import numpy as np

np.random.seed(0)
rng = pd.date_range('2015-02-24', periods=5, freq='T')
df = pd.DataFrame({ 'Val' : np.random.randn(len(rng)) }, index=rng)

print (df)
# Output:
#                           Val
# 2015-02-24 00:00:00  1.764052
# 2015-02-24 00:01:00  0.400157
# 2015-02-24 00:02:00  0.978738
# 2015-02-24 00:03:00  2.240893
# 2015-02-24 00:04:00  1.867558
```

`Offset-aliases` pour le paramètre `freq` dans `date_range` :

```
Alias     Description
B         business day frequency
C         custom business day frequency (experimental)
```

```
D          calendar day frequency
W          weekly frequency
M          month end frequency
BM         business month end frequency
CBM        custom business month end frequency
MS         month start frequency
BMS        business month start frequency
CBMS       custom business month start frequency
Q          quarter end frequency
BQ         business quarter endfrequency
QS         quarter start frequency
BQS        business quarter start frequency
A          year end frequency
BA         business year end frequency
AS         year start frequency
BAS        business year start frequency
BH         business hour frequency
H          hourly frequency
T, min     minutely frequency
S          secondly frequency
L, ms      milliseconds
U, us      microseconds
N          nanoseconds
```

Créer un exemple de DataFrame avec MultiIndex

```
import pandas as pd
import numpy as np
```

Utiliser `from_tuples` :

```
np.random.seed(0)
tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                     'foo', 'foo', 'qux', 'qux'],
                      ['one', 'two', 'one', 'two',
                       'one', 'two', 'one', 'two']]))

idx = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
```

Utiliser `from_product` :

```
idx = pd.MultiIndex.from_product([['bar', 'baz', 'foo', 'qux'],['one','two']])
```

Ensuite, utilisez ce MultiIndex:

```
df = pd.DataFrame(np.random.randn(8, 2), index=idx, columns=['A', 'B'])
print (df)
                     A         B
first second
bar   one     1.764052  0.400157
      two     0.978738  2.240893
baz   one     1.867558 -0.977278
      two     0.950088 -0.151357
foo   one    -0.103219  0.410599
      two     0.144044  1.454274
```

```
qux   one      0.761038  0.121675
      two      0.443863  0.333674
```

Enregistrer et charger un DataFrame au format pickle (.plk)

```
import pandas as pd

# Save dataframe to pickled pandas object
df.to_pickle(file_name) # where to save it usually as a .plk

# Load dataframe from pickled pandas object
df= pd.read_pickle(file_name)
```

Créer un DataFrame à partir d'une liste de dictionnaires

Un DataFrame peut être créé à partir d'une liste de dictionnaires. Les clés sont utilisées comme noms de colonne.

```
import pandas as pd
L = [{'Name': 'John', 'Last Name': 'Smith'},
         {'Name': 'Mary', 'Last Name': 'Wood'}]
pd.DataFrame(L)
# Output:  Last Name  Name
# 0     Smith  John
# 1      Wood  Mary
```

Les valeurs manquantes sont remplies avec `NaN` s

```
L = [{'Name': 'John', 'Last Name': 'Smith', 'Age': 37},
     {'Name': 'Mary', 'Last Name': 'Wood'}]
pd.DataFrame(L)
# Output:     Age Last Name  Name
#          0   37     Smith  John
#          1  NaN      Wood  Mary
```

Lire Création de DataFrames en ligne: https://riptutorial.com/fr/pandas/topic/1595/creation-de-dataframes

Chapitre 6: Données catégoriques

Introduction

Les catégories sont un type de données pandas, qui correspond à des variables catégorielles dans les statistiques: une variable qui ne peut prendre qu'un nombre limité et généralement fixe de valeurs possibles (catégories; niveaux dans R). Les exemples sont le sexe, la classe sociale, les groupes sanguins, les affiliations par pays, le temps d'observation ou les évaluations via les échelles de Likert. Source: Pandas Docs

Examples

Création d'objet

```
In [188]: s = pd.Series(["a","b","c","a","c"], dtype="category")

In [189]: s
Out[189]:
0    a
1    b
2    c
3    a
4    c
dtype: category
Categories (3, object): [a, b, c]

In [190]: df = pd.DataFrame({"A":["a","b","c","a", "c"]})

In [191]: df["B"] = df["A"].astype('category')

In [192]: df["C"] = pd.Categorical(df["A"])

In [193]: df
Out[193]:
   A  B  C
0  a  a  a
1  b  b  b
2  c  c  c
3  a  a  a
4  c  c  c

In [194]: df.dtypes
Out[194]:
A      object
B    category
C    category
dtype: object
```

Création de jeux de données aléatoires volumineux

```
In [1]: import pandas as pd
        import numpy as np
```

```
In [2]: df = pd.DataFrame(np.random.choice(['foo','bar','baz'], size=(100000,3)))
        df = df.apply(lambda col: col.astype('category'))

In [3]: df.head()
Out[3]:
     0    1    2
0  bar  foo  baz
1  baz  bar  baz
2  foo  foo  bar
3  bar  baz  baz
4  foo  bar  baz

In [4]: df.dtypes
Out[4]:
0    category
1    category
2    category
dtype: object

In [5]: df.shape
Out[5]: (100000, 3)
```

Lire Données catégoriques en ligne: https://riptutorial.com/fr/pandas/topic/3887/donnees-categoriques

Chapitre 7: Données décalées et décalées

Examples

Décalage ou décalage de valeurs dans un dataframe

```
import pandas as pd

df = pd.DataFrame({'eggs': [1,2,4,8,], 'chickens': [0,1,2,4,]})

df

#    chickens  eggs
# 0         0     1
# 1         1     2
# 2         2     4
# 3         4     8

df.shift()

#    chickens  eggs
# 0       NaN   NaN
# 1       0.0   1.0
# 2       1.0   2.0
# 3       2.0   4.0

df.shift(-2)

#    chickens  eggs
# 0       2.0   4.0
# 1       4.0   8.0
# 2       NaN   NaN
# 3       NaN   NaN

df['eggs'].shift(1) - df['chickens']

# 0    NaN
# 1    0.0
# 2    0.0
# 3    0.0
```

Le premier argument de `.shift()` est des `periods` , le nombre d'espaces pour déplacer les données. Si non spécifié, la valeur par défaut est `1` .

Lire Données décalées et décalées en ligne: https://riptutorial.com/fr/pandas/topic/7554/donnees-decalees-et-decalees

Chapitre 8: Données dupliquées

Examples

Sélectionnez dupliqué

Si nécessaire, définissez la valeur `0` sur la colonne `B` , où dans la colonne `A` les données dupliquées créent d'abord le masque par `Series.duplicated` , puis utilisent `DataFrame.ix` ou `Series.mask` :

```
In [224]: df = pd.DataFrame({'A':[1,2,3,3,2],
     ...:                    'B':[1,7,3,0,8]})

In [225]: mask = df.A.duplicated(keep=False)

In [226]: mask
Out[226]:
0    False
1     True
2     True
3     True
4     True
Name: A, dtype: bool

In [227]: df.ix[mask, 'B'] = 0

In [228]: df['C'] = df.A.mask(mask, 0)

In [229]: df
Out[229]:
   A  B  C
0  1  1  1
1  2  0  0
2  3  0  0
3  3  0  0
4  2  0  0
```

Si besoin inverser masque utiliser `~` :

```
In [230]: df['C'] = df.A.mask(~mask, 0)

In [231]: df
Out[231]:
   A  B  C
0  1  1  0
1  2  0  2
2  3  0  3
3  3  0  3
4  2  0  2
```

Drop dupliqué

Utilisez `drop_duplicates` :

```
In [216]: df = pd.DataFrame({'A':[1,2,3,3,2],
     ...:                    'B':[1,7,3,0,8]})

In [217]: df
Out[217]:
   A  B
0  1  1
1  2  7
2  3  3
3  3  0
4  2  8

# keep only the last value
In [218]: df.drop_duplicates(subset=['A'], keep='last')
Out[218]:
   A  B
0  1  1
3  3  0
4  2  8

# keep only the first value, default value
In [219]: df.drop_duplicates(subset=['A'], keep='first')
Out[219]:
   A  B
0  1  1
1  2  7
2  3  3

# drop all duplicated values
In [220]: df.drop_duplicates(subset=['A'], keep=False)
Out[220]:
   A  B
0  1  1
```

Lorsque vous ne souhaitez pas obtenir une copie d'un bloc de données, mais modifier celle existante:

```
In [221]: df = pd.DataFrame({'A':[1,2,3,3,2],
     ...:                    'B':[1,7,3,0,8]})

In [222]: df.drop_duplicates(subset=['A'], inplace=True)

In [223]: df
Out[223]:
   A  B
0  1  1
1  2  7
2  3  3
```

Compter et obtenir des éléments uniques

Nombre d'éléments uniques dans une série:

```
In [1]: id_numbers = pd.Series([111, 112, 112, 114, 115, 118, 114, 118, 112])
In [2]: id_numbers.nunique()
Out[2]: 5
```

Obtenez des éléments uniques dans une série:

```
In [3]: id_numbers.unique()
Out[3]: array([111, 112, 114, 115, 118], dtype=int64)

In [4]: df = pd.DataFrame({'Group': list('ABAABABAAB'),
                           'ID': [1, 1, 2, 3, 3, 2, 1, 2, 1, 3]})

In [5]: df
Out[5]:
  Group  ID
0     A   1
1     B   1
2     A   2
3     A   3
4     B   3
5     A   2
6     B   1
7     A   2
8     A   1
9     B   3
```

Nombre d'éléments uniques dans chaque groupe:

```
In [6]: df.groupby('Group')['ID'].nunique()
Out[6]:
Group
A    3
B    2
Name: ID, dtype: int64
```

Obtenez des éléments uniques dans chaque groupe:

```
In [7]: df.groupby('Group')['ID'].unique()
Out[7]:
Group
A    [1, 2, 3]
B       [1, 3]
Name: ID, dtype: object
```

Obtenez des valeurs uniques dans une colonne.

```
In [15]: df = pd.DataFrame({"A":[1,1,2,3,1,1],"B":[5,4,3,4,6,7]})

In [21]: df
Out[21]:
   A  B
0  1  5
1  1  4
2  2  3
3  3  4
4  1  6
5  1  7
```

Pour obtenir des valeurs uniques dans les colonnes A et B.

```
In [22]: df["A"].unique()
Out[22]: array([1, 2, 3])

In [23]: df["B"].unique()
Out[23]: array([5, 4, 3, 6, 7])
```

Pour obtenir les valeurs uniques de la colonne A en tant que liste (notez que `unique()` peut être utilisé de deux manières légèrement différentes)

```
In [24]: pd.unique(df['A']).tolist()
Out[24]: [1, 2, 3]
```

Voici un exemple plus complexe. Disons que nous voulons trouver les valeurs uniques de la colonne 'B' où 'A' est égal à 1.

D'abord, introduisons un doublon pour voir comment cela fonctionne. Remplaçons le 6 dans la ligne '4', la colonne 'B' avec un 4:

```
In [24]: df.loc['4', 'B'] = 4
Out[24]:
   A  B
0  1  5
1  1  4
2  2  3
3  3  4
4  1  4
5  1  7
```

Sélectionnez maintenant les données:

```
In [25]: pd.unique(df[df['A'] == 1 ]['B']).tolist()
Out[25]: [5, 4, 7]
```

Cela peut être décomposé en pensant au DataFrame interne en premier:

```
df['A'] == 1
```

Cela trouve des valeurs dans la colonne A qui sont égales à 1 et leur applique True ou False. Nous pouvons alors l'utiliser pour sélectionner les valeurs de la colonne 'B' du DataFrame (la sélection externe DataFrame)

Pour comparaison, voici la liste si nous n'utilisons pas unique. Il récupère chaque valeur dans la colonne 'B' où la colonne 'A' est 1

```
In [26]: df[df['A'] == 1]['B'].tolist()
Out[26]: [5, 4, 4, 7]
```

Lire Données dupliquées en ligne: https://riptutorial.com/fr/pandas/topic/2082/donnees-dupliquees

Chapitre 9: Données manquantes

Remarques

Devrions-nous inclure le `ffill` et le `bfill` non documentés?

Examples

Remplir les valeurs manquantes

```
In [11]: df = pd.DataFrame([[1, 2, None, 3], [4, None, 5, 6],
                            [7, 8, 9, 10], [None, None, None, None]])

Out[11]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  NaN  5.0   6.0
2  7.0  8.0  9.0  10.0
3  NaN  NaN  NaN   NaN
```

Remplir les valeurs manquantes avec une seule valeur:

```
In [12]: df.fillna(0)
Out[12]:
     0    1    2     3
0  1.0  2.0  0.0   3.0
1  4.0  0.0  5.0   6.0
2  7.0  8.0  9.0  10.0
3  0.0  0.0  0.0   0.0
```

Cela retourne un nouveau DataFrame. Si vous souhaitez modifier le DataFrame d'origine, utilisez le paramètre `inplace` (`df.fillna(0, inplace=True)`) ou attribuez-le au fichier de données d'origine (`df = df.fillna(0)`).

Remplissez les valeurs manquantes avec les précédentes:

```
In [13]: df.fillna(method='pad')  # this is equivalent to both method='ffill' and .ffill()
Out[13]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  2.0  5.0   6.0
2  7.0  8.0  9.0  10.0
3  7.0  8.0  9.0  10.0
```

Remplissez avec les suivants:

```
In [14]: df.fillna(method='bfill')  # this is equivalent to .bfill()
Out[14]:
     0    1    2     3
0  1.0  2.0  5.0   3.0
1  4.0  8.0  5.0   6.0
2  7.0  8.0  9.0  10.0
3  NaN  NaN  NaN   NaN
```

Remplir à l'aide d'un autre DataFrame:

```
In [15]: df2 = pd.DataFrame(np.arange(100, 116).reshape(4, 4))
         df2
Out[15]:
     0    1    2    3
0  100  101  102  103
1  104  105  106  107
2  108  109  110  111
3  112  113  114  115

In [16]: df.fillna(df2) #  takes the corresponding cells in df2 to fill df
Out[16]:
       0      1      2      3
0    1.0    2.0  102.0    3.0
1    4.0  105.0    5.0    6.0
2    7.0    8.0    9.0   10.0
3  112.0  113.0  114.0  115.0
```

Supprimer les valeurs manquantes

Lors de la création d'un DataFrame `None` (valeur manquante de python) est converti en `NaN` (valeur manquante des pandas):

```
In [11]: df = pd.DataFrame([[1, 2, None, 3], [4, None, 5, 6],
                            [7, 8, 9, 10], [None, None, None, None]])

Out[11]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  NaN  5.0   6.0
2  7.0  8.0  9.0  10.0
3  NaN  NaN  NaN   NaN
```

Supprimer des lignes si au moins une colonne a une valeur manquante

```
In [12]: df.dropna()
Out[12]:
     0    1    2     3
2  7.0  8.0  9.0  10.0
```

Cela retourne un nouveau DataFrame. Si vous souhaitez modifier le DataFrame d'origine, utilisez le paramètre `inplace` (`df.dropna(inplace=True)`) ou attribuez-le à DataFrame d'origine (`df =`

`df.dropna()`).

Supprimer des lignes si toutes les valeurs de cette ligne sont manquantes

```
In [13]: df.dropna(how='all')
Out[13]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  NaN  5.0   6.0
2  7.0  8.0  9.0  10.0
```

Supprimez les *colonnes* qui n'ont pas au moins 3 valeurs non manquantes

```
In [14]: df.dropna(axis=1, thresh=3)
Out[14]:
     0     3
0  1.0   3.0
1  4.0   6.0
2  7.0  10.0
3  NaN   NaN
```

Interpolation

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A':[1,2,np.nan,3,np.nan],
                   'B':[1.2,7,3,0,8]})

df['C'] = df.A.interpolate()
df['D'] = df.A.interpolate(method='spline', order=1)

print (df)
     A    B    C         D
0  1.0  1.2  1.0  1.000000
1  2.0  7.0  2.0  2.000000
2  NaN  3.0  2.5  2.428571
3  3.0  0.0  3.0  3.000000
4  NaN  8.0  3.0  3.714286
```

Vérification des valeurs manquantes

Afin de vérifier si une valeur est NaN, les fonctions `isnull()` ou `notnull()` peuvent être utilisées.

```
In [1]: import numpy as np
In [2]: import pandas as pd
In [3]: ser = pd.Series([1, 2, np.nan, 4])
In [4]: pd.isnull(ser)
Out[4]:
```

```
0    False
1    False
2     True
3    False
dtype: bool
```

Notez que `np.nan == np.nan` renvoie False, vous devez donc éviter la comparaison avec np.nan:

```
In [5]: ser == np.nan
Out[5]:
0    False
1    False
2    False
3    False
dtype: bool
```

Les deux fonctions sont également définies en tant que méthodes sur les séries et les DataFrames.

```
In [6]: ser.isnull()
Out[6]:
0    False
1    False
2     True
3    False
dtype: bool
```

Test sur des DataFrames:

```
In [7]: df = pd.DataFrame({'A': [1, np.nan, 3], 'B': [np.nan, 5, 6]})
In [8]: print(df)
Out[8]:
     A    B
0  1.0  NaN
1  NaN  5.0
2  3.0  6.0

In [9]: df.isnull()  # If the value is NaN, returns True.
Out[9]:
       A      B
0  False   True
1   True  False
2  False  False

In [10]: df.notnull()  # Opposite of .isnull(). If the value is not NaN, returns True.
Out[10]:
       A      B
0   True  False
1  False   True
2   True   True
```

Lire Données manquantes en ligne: https://riptutorial.com/fr/pandas/topic/1896/donnees-manquantes

Chapitre 10: Enregistrer les données pandas dans un fichier csv

Paramètres

Paramètre	La description
path_or_buf	handle de chaîne ou de fichier, par défaut Aucun Chemin ou objet de fichier, si aucun est fourni, le résultat est renvoyé sous forme de chaîne.
sep	character, default ',' Délimiteur de champ pour le fichier de sortie.
na_rep	string, default '' Représentation des données manquantes
float_format	string, default None Chaîne de format pour les nombres à virgule flottante
colonnes	séquence, colonnes facultatives à écrire
entête	booléen ou liste de chaîne, par défaut True Ecrivez les noms de colonne. Si une liste de chaîne est donnée, elle est supposée être un alias pour les noms de colonne
indice	booléen, par défaut True Write noms de lignes (index)
index_label	string ou sequence, ou False, default Aucun Etiquette de colonne pour les colonnes d'index si vous le souhaitez. Si None est donné et que header et index sont True, les noms d'index sont utilisés. Une séquence doit être donnée si le DataFrame utilise MultiIndex. Si False, n'imprimez pas de champs pour les noms d'index. Utilisez index_label = False pour importer plus facilement dans R
nanRep	Aucun déprécié, utilisez na_rep
mode	str Mode d'écriture Python, par défaut 'w'
codage	string, optionnel Chaîne représentant l'encodage à utiliser dans le fichier de sortie, par défaut, 'ascii' sur Python 2 et 'utf-8' sur Python 3.
compression	string, facultatif une chaîne représentant la compression à utiliser dans le fichier de sortie, les valeurs autorisées sont 'gzip', 'bz2', 'xz', utilisé uniquement lorsque le premier argument est un nom de fichier
line_terminator	string, default 'n' Caractère de nouvelle ligne ou séquence de caractères à utiliser dans le fichier de sortie
en citant	constante optionnelle du module csv par défaut à csv.QUOTE_MINIMAL

Paramètre	La description
quotechar	string (length 1), caractère par défaut "" utilisé pour citer les champs
double citation	booléen, par défaut True Control citant quotechar dans un champ
escapechar	chaîne (longueur 1), par défaut Aucun caractère utilisé pour échapper à sépare et quotechar, le cas échéant
taille	lignes int ou None à écrire à la fois
tupleize_cols	booléen, par défaut False écrit des colonnes multi_index comme liste de tuples (si True) ou nouveau (format développé) si False)
format de date	string, default None Chaîne de formatage pour les objets datetime
décimal	string, default '.' Caractère reconnu comme séparateur décimal. Par exemple, utiliser "," pour les données européennes

Examples

Créez un DataFrame aléatoire et écrivez dans .csv

Créez un simple DataFrame.

```
import numpy as np
import pandas as pd

# Set the seed so that the numbers can be reproduced.
np.random.seed(0)

df = pd.DataFrame(np.random.randn(5, 3), columns=list('ABC'))

# Another way to set column names is
"columns=['column_1_name','column_2_name','column_3_name']"

df

      A         B         C
0  1.764052  0.400157  0.978738
1  2.240893  1.867558 -0.977278
2  0.950088 -0.151357 -0.103219
3  0.410599  0.144044  1.454274
4  0.761038  0.121675  0.443863
```

Maintenant, écrivez dans un fichier CSV:

```
df.to_csv('example.csv', index=False)
```

Contenu de exemple.csv:

```
A,B,C
```

```
1.76405234597,0.400157208367,0.978737984106
2.2408931992,1.86755799015,-0.977277879876
0.950088417526,-0.151357208298,-0.103218851794
0.410598501938,0.144043571161,1.45427350696
0.761037725147,0.121675016493,0.443863232745
```

Notez que nous spécifions `index=False` pour que les index générés automatiquement (n ° de ligne 0,1,2,3,4) ne soient pas inclus dans le fichier CSV. Incluez-le si vous avez besoin de la colonne d'index, comme ceci:

```
df.to_csv('example.csv', index=True)  # Or just leave off the index param; default is True
```

Contenu de exemple.csv:

```
,A,B,C
0,1.76405234597,0.400157208367,0.978737984106
1,2.2408931992,1.86755799015,-0.977277879876
2,0.950088417526,-0.151357208298,-0.103218851794
3,0.410598501938,0.144043571161,1.45427350696
4,0.761037725147,0.121675016493,0.443863232745
```

Notez également que vous pouvez supprimer l'en-tête s'il n'est pas nécessaire avec `header=False` . C'est la sortie la plus simple:

```
df.to_csv('example.csv', index=False, header=False)
```

Contenu de exemple.csv:

```
1.76405234597,0.400157208367,0.978737984106
2.2408931992,1.86755799015,-0.977277879876
0.950088417526,-0.151357208298,-0.103218851794
0.410598501938,0.144043571161,1.45427350696
0.761037725147,0.121675016493,0.443863232745
```

Le séparateur peut être défini par `sep=` argument, bien que le séparateur standard pour les fichiers csv soit `','` .

```
df.to_csv('example.csv', index=False, header=False, sep='\t')
```

```
1.76405234597    0.400157208367    0.978737984106
2.2408931992    1.86755799015    -0.977277879876
0.950088417526    -0.151357208298    -0.103218851794
0.410598501938    0.144043571161    1.45427350696
0.761037725147    0.121675016493    0.443863232745
```

Enregistrer Pandas DataFrame de la liste aux dicts à csv sans index et avec encodage des données

```
import pandas as pd
data = [
    {'name': 'Daniel', 'country': 'Uganda'},
```

```
    {'name': 'Yao', 'country': 'China'},
    {'name': 'James', 'country': 'Colombia'},
]
df = pd.DataFrame(data)
filename = 'people.csv'
df.to_csv(filename, index=False, encoding='utf-8')
```

Lire Enregistrer les données pandas dans un fichier csv en ligne:
https://riptutorial.com/fr/pandas/topic/1558/enregistrer-les-donnees-pandas-dans-un-fichier-csv

Chapitre 11: Faire jouer les Pandas avec les types de données Python natifs

Examples

Déplacement de données hors de pandas vers des structures de données natives Python et Numpy

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': ['a', 'b', 'c'],
                           'D': [True, False, True]})

In [2]: df
Out[2]:
   A    B  C      D
0  1  1.0  a   True
1  2  2.0  b  False
2  3  3.0  c   True
```

Obtenir une liste de python à partir d'une série:

```
In [3]: df['A'].tolist()
Out[3]: [1, 2, 3]
```

Les DataFrames n'ont pas de méthode `tolist()` . Son essai entraîne une erreur d'attribut:

```
In [4]: df.tolist()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-fc6763af1ff7> in <module>()
----> 1 df.tolist()

//anaconda/lib/python2.7/site-packages/pandas/core/generic.pyc in __getattr__(self, name)
   2742             if name in self._info_axis:
   2743                 return self[name]
-> 2744             return object.__getattribute__(self, name)
   2745
   2746     def __setattr__(self, name, value):

AttributeError: 'DataFrame' object has no attribute 'tolist'
```

Obtenir un tableau numpy d'une série:

```
In [5]: df['B'].values
Out[5]: array([ 1.,  2.,  3.])
```

Vous pouvez également obtenir un tableau des colonnes sous forme de tableaux numpy individuels à partir d'un cadre de données complet:

```
In [6]: df.values
```

```
Out[6]:
array([[1, 1.0, 'a', True],
       [2, 2.0, 'b', False],
       [3, 3.0, 'c', True]], dtype=object)
```

Obtenir un dictionnaire à partir d'une série (utilise l'index comme clés):

```
In [7]: df['C'].to_dict()
Out[7]: {0: 'a', 1: 'b', 2: 'c'}
```

Vous pouvez également récupérer l'intégralité de DataFrame en tant que dictionnaire:

```
In [8]: df.to_dict()
Out[8]:
{'A': {0: 1, 1: 2, 2: 3},
 'B': {0: 1.0, 1: 2.0, 2: 3.0},
 'C': {0: 'a', 1: 'b', 2: 'c'},
 'D': {0: True, 1: False, 2: True}}
```

La méthode `to_dict` a quelques paramètres différents pour ajuster le format des dictionnaires. Pour obtenir une liste de dicts pour chaque ligne:

```
In [9]: df.to_dict('records')
Out[9]:
[{'A': 1, 'B': 1.0, 'C': 'a', 'D': True},
 {'A': 2, 'B': 2.0, 'C': 'b', 'D': False},
 {'A': 3, 'B': 3.0, 'C': 'c', 'D': True}]
```

Voir la documentation pour la liste complète des options disponibles pour créer des dictionnaires.

Lire Faire jouer les Pandas avec les types de données Python natifs en ligne: https://riptutorial.com/fr/pandas/topic/8008/faire-jouer-les-pandas-avec-les-types-de-donnees-python-natifs

Chapitre 12: Fusionner, rejoindre et concaténer

Syntaxe

- Trame de données. **merge** (*right, how = 'inner', on = None, left_on = None, right_on = Aucun, left_index = False, right_index = False, sort = False, suffixes = ('_ x', '_y'), copy = True, indicateur = Faux*)
- Fusionner des objets DataFrame en effectuant une opération de jointure de style base de données par colonnes ou index.
- Si vous associez des colonnes sur des colonnes, les index DataFrame seront ignorés. Sinon, si vous joignez des index sur des index ou des index sur une ou plusieurs colonnes, l'index sera transmis.

Paramètres

Paramètres	Explication
droite	Trame de données
Comment	{'gauche', 'droite', 'extérieur', 'intérieur'}, défaut 'intérieur'
à gauche sur	label ou list, ou tableau-like. Noms de champs sur lesquels se joindre dans DataFrame gauche. Peut être un vecteur ou une liste de vecteurs de la longueur du DataFrame pour utiliser un vecteur particulier comme clé de jointure au lieu de colonnes
à droite	label ou list, ou tableau-like. Noms de champs sur lesquels se joindre à droite DataFrame ou vecteur / liste de vecteurs par left_on docs
left_index	booléen, par défaut False. Utilisez l'index du DataFrame de gauche comme clé (s) de jointure. S'il s'agit d'un MultiIndex, le nombre de clés dans l'autre DataFrame (l'index ou un nombre de colonnes) doit correspondre au nombre de niveaux.
right_index	booléen, par défaut False. Utilisez l'index du bon DataFrame comme clé de jointure. Même mise en garde que left_index
Trier	booléen, Fals par défaut. Trier les clés de jointure lexicographiquement dans le résultat DataFrame
suffixes	Séquence de 2 longueurs (tuple, liste, ...). Suffixe à appliquer aux noms de colonnes qui se chevauchent respectivement à gauche et à droite

Paramètres	Explication
copie	booléen, par défaut True. Si la valeur est False, ne copiez pas inutilement des données
indicateur	booléen ou chaîne, par défaut False. Si True, ajoute une colonne à la sortie DataFrame appelée «_merge» avec des informations sur la source de chaque ligne. Si chaîne, une colonne avec des informations sur la source de chaque ligne sera ajoutée à la sortie DataFrame, et la colonne sera nommée valeur de la chaîne. La colonne d'information est de type catégorique et prend la valeur «left_only» pour les observations dont la clé de fusion n'apparaît que dans le cadre «left» DataFrame, «right_only» pour les observations dont la clé de fusion n'apparaît que dans «right» DataFrame et «both» si La clé de fusion de l'observation se trouve dans les deux.

Examples

Fusionner

Par exemple, deux tables sont données,

T1

```
id    x        y
8    42        1.9
9    30        1.9
```

T2

```
id    signal
8    55
8    56
8    59
9    57
9    58
9    60
```

L'objectif est d'obtenir la nouvelle table T3:

```
id    x        y        s1        s2        s3
8    42        1.9        55        56        58
9    30        1.9        57        58        60
```

Ce qui consiste à créer les colonnes `s1` , `s2` et `s3` , chacune correspondant à une ligne (le nombre de lignes par `id` est toujours fixe et égal à 3)

En appliquant la `join` (qui prend un argument facultatif qui peut être une colonne ou plusieurs noms de colonne, ce qui spécifie que le DataFrame transmis doit être aligné sur cette colonne dans le DataFrame). La solution peut donc être la suivante:

df = df1.merge (df2.groupby ('id') ['signal']. apply (lambda x: x.reset_index (drop = True)). unstack (). reset_index ())

```
df
Out[63]:
   id   x    y   0   1   2
0   8  42  1.9  55  56  59
1   9  30  1.9  57  58  60
```

Si je les sépare:

```
df2t = df2.groupby('id')['signal'].apply(lambda x:
x.reset_index(drop=True)).unstack().reset_index()

df2t
Out[59]:
   id   0   1   2
0   8  55  56  59
1   9  57  58  60

df = df1.merge(df2t)

df
Out[61]:
   id   x    y   0   1   2
0   8  42  1.9  55  56  59
1   9  30  1.9  57  58  60
```

Fusion de deux DataFrames

```
In [1]: df1 = pd.DataFrame({'x': [1, 2, 3], 'y': ['a', 'b', 'c']})

In [2]: df2 = pd.DataFrame({'y': ['b', 'c', 'd'], 'z': [4, 5, 6]})

In [3]: df1
Out[3]:
   x  y
0  1  a
1  2  b
2  3  c

In [4]: df2
Out[4]:
   y  z
0  b  4
1  c  5
2  d  6
```

Jointure interne:

Utilise l'intersection des clés de deux DataFrames.

```
In [5]: df1.merge(df2) # by default, it does an inner join on the common column(s)
```

```
Out[5]:
   x  y  z
0  2  b  4
1  3  c  5
```

Vous pouvez également spécifier l'intersection des clés à partir de deux cadres de données.

```
In [5]: merged_inner = pd.merge(left=df1, right=df2, left_on='y', right_on='y')
Out[5]:
   x  y  z
0  2  b  4
1  3  c  5
```

Jointure externe:

Utilise l'union des clés de deux DataFrames.

```
In [6]: df1.merge(df2, how='outer')
Out[6]:
     x  y    z
0  1.0  a  NaN
1  2.0  b  4.0
2  3.0  c  5.0
3  NaN  d  6.0
```

Joint gauche:

Utilise uniquement les clés de gauche DataFrame.

```
In [7]: df1.merge(df2, how='left')
Out[7]:
   x  y    z
0  1  a  NaN
1  2  b  4.0
2  3  c  5.0
```

Droit rejoindre

Utilise uniquement les clés du droit DataFrame.

```
In [8]: df1.merge(df2, how='right')
Out[8]:
     x  y  z
0  2.0  b  4
1  3.0  c  5
2  NaN  d  6
```

Fusion / concaténation / jonction de plusieurs blocs de données (horizontalement et verticalement)

générer des exemples de trames de données:

```
In [57]: df3 = pd.DataFrame({'col1':[211,212,213], 'col2': [221,222,223]})

In [58]: df1 = pd.DataFrame({'col1':[11,12,13], 'col2': [21,22,23]})

In [59]: df2 = pd.DataFrame({'col1':[111,112,113], 'col2': [121,122,123]})

In [60]: df3 = pd.DataFrame({'col1':[211,212,213], 'col2': [221,222,223]})

In [61]: df1
Out[61]:
   col1  col2
0    11    21
1    12    22
2    13    23

In [62]: df2
Out[62]:
   col1  col2
0   111   121
1   112   122
2   113   123

In [63]: df3
Out[63]:
   col1  col2
0   211   221
1   212   222
2   213   223
```

fusionner / joindre / concaténer des trames de données [df1, df2, df3] verticalement - ajouter des lignes

```
In [64]: pd.concat([df1,df2,df3], ignore_index=True)
Out[64]:
   col1  col2
0    11    21
1    12    22
2    13    23
3   111   121
4   112   122
5   113   123
6   211   221
7   212   222
8   213   223
```

fusionner / joindre / concaténer des blocs de données horizontalement (alignement par index):

```
In [65]: pd.concat([df1,df2,df3], axis=1)
Out[65]:
   col1  col2  col1  col2  col1  col2
0    11    21   111   121   211   221
```

```
1    12    22   112   122   212   222
2    13    23   113   123   213   223
```

Fusionner, rejoindre et concat

La fusion des noms de clés est la même

```
pd.merge(df1, df2, on='key')
```

La fusion des noms de clés est différente

```
pd.merge(df1, df2, left_on='l_key', right_on='r_key')
```

Différents types d'adhésion

```
pd.merge(df1, df2, on='key', how='left')
```

Fusion sur plusieurs clés

```
pd.merge(df1, df2, on=['key1', 'key2'])
```

Traitement des colonnes superposées

```
pd.merge(df1, df2, on='key', suffixes=('_left', '_right'))
```

Utilisation de l'index de lignes au lieu de fusionner des clés

```
pd.merge(df1, df2, right_index=True, left_index=True)
```

Évitez d'utiliser la syntaxe `.join` car elle donne une exception pour les colonnes qui se chevauchent

Fusion sur l'index de dataframe gauche et la colonne dataframe droite

```
pd.merge(df1, df2, right_index=True, left_on='l_key')
```

Concentrer les dataframes

Collé verticalement

```
pd.concat([df1, df2, df3], axis=0)
```

Collé horizontalement

```
pd.concat([df1, df2, df3], axis=1)
```

Quelle est la différence entre rejoindre et fusionner

Considérons les données à `left` et à `right`

```
left = pd.DataFrame([['a', 1], ['b', 2]], list('XY'), list('AB'))
left

   A  B
X  a  1
Y  b  2
```

```
right = pd.DataFrame([['a', 3], ['b', 4]], list('XY'), list('AC'))
right

   A  C
X  a  3
Y  b  4
```

join

Pensez à `join` que de vouloir combiner pour dataframes en fonction de leurs indices respectifs. S'il y a des colonnes qui se chevauchent, `join` voudra que vous ajoutiez un suffixe au nom de la colonne qui se chevauchent à partir du dataframe gauche. Nos deux cadres de données ont un nom de colonne qui se chevauche `A`

```
left.join(right, lsuffix='_')

  A_  B  A  C
X  a  1  a  3
Y  b  2  b  4
```

Notez que l'index est conservé et que nous avons 4 colonnes. 2 colonnes de `left` et 2 de `right` .

Si les index ne sont pas alignés

```
left.join(right.reset_index(), lsuffix='_', how='outer')

    A_    B index    A    C
0  NaN  NaN     X    a  3.0
1  NaN  NaN     Y    b  4.0
X    a  1.0   NaN  NaN  NaN
Y    b  2.0   NaN  NaN  NaN
```

J'ai utilisé une jointure externe pour mieux illustrer ce point. Si les index ne sont pas alignés, le résultat sera l'union des index.

Nous pouvons dire à `join` d'utiliser une colonne spécifique dans le dataframe de gauche à utiliser comme clé de jointure, mais elle utilisera toujours l'index depuis la droite.

```
left.reset_index().join(right, on='index', lsuffix='_')

  index A_  B  A  C
0     X  a  1  a  3
```

```
1     Y  b  2  b  4
```

merge

Pensez à `merge` en alignant sur des colonnes. Par défaut, la `merge` recherche les colonnes qui se chevauchent dans lesquelles fusionner. `merge` donne un meilleur contrôle sur les clés de fusion en permettant à l'utilisateur de spécifier un sous - ensemble des colonnes qui se chevauchent à utiliser avec le paramètre `on` ou pour permettre séparément dont la spécification des colonnes à gauche et les colonnes sur le droit de fusionner par.

`merge` renverra un dataframe combiné dans lequel l'index sera détruit.

Cet exemple simple recherche la colonne qui se chevauche pour être `'A'` et combine en fonction de celle-ci.

```
left.merge(right)

   A  B  C
0  a  1  3
1  b  2  4
```

Notez que l'index est `[0, 1]` et non plus `['X', 'Y']`

Vous pouvez spécifier explicitement que vous fusionnez sur l'indice avec le `left_index` ou `right_index` paramter

```
left.merge(right, left_index=True, right_index=True, suffixes=['_', ''])

  A_  B  A  C
X  a  1  a  3
Y  b  2  b  4
```

Et cela ressemble exactement à l'exemple de `join` ci-dessus.

Lire Fusionner, rejoindre et concaténer en ligne:
https://riptutorial.com/fr/pandas/topic/1966/fusionner--rejoindre-et-concatener

Chapitre 13: Gotchas de pandas

Remarques

Gotcha en général est une construction bien que documentée, mais pas intuitive. Les Gotchas produisent des résultats normalement inattendus en raison de leur caractère contre-intuitif.

Le paquet Pandas a plusieurs pièges, qui peuvent induire en erreur quelqu'un, qui n'en a pas connaissance, et certains d'entre eux sont présentés sur cette page de documentation.

Examples

Détecter les valeurs manquantes avec np.nan

Si vous voulez détecter les manquements avec

```
df=pd.DataFrame({'col':[1,np.nan]})
df==np.nan
```

vous obtiendrez le résultat suivant:

```
col
0    False
1    False
```

C'est parce que la comparaison des valeurs manquantes avec quelque chose donne lieu à un False - au lieu de cela, vous devez utiliser

```
df=pd.DataFrame({'col':[1,np.nan]})
df.isnull()
```

qui se traduit par:

```
col
0    False
1    True
```

Entier et NA

Les pandas ne prennent pas en charge les attributs manquants du type entier. Par exemple si vous avez des manques dans la colonne de notation:

```
df= pd.read_csv("data.csv", dtype={'grade': int})
error: Integer column has NA values
```

Dans ce cas, vous devez simplement utiliser float au lieu de nombres entiers ou définir le type

d'objet.

Alignement automatique des données (comportement indexé)

Si vous souhaitez ajouter une série de valeurs [1,2] à la colonne de dataframe df, vous obtiendrez NaN:

```
import pandas as pd

series=pd.Series([1,2])
df=pd.DataFrame(index=[3,4])
df['col']=series
df

   col
3    NaN
4    NaN
```

car la définition d'une nouvelle colonne aligne automatiquement les données par l'index et vos valeurs 1 et 2 obtiendront les index 0 et 1 et non 3 et 4 comme dans votre bloc de données:

```
df=pd.DataFrame(index=[1,2])
df['col']=series
df

   col
1      2.0
2      NaN
```

Si vous voulez ignorer l'index, vous devez définir les valeurs à la fin:

```
df['col']=series.values

   col
3    1
4    2
```

Lire Gotchas de pandas en ligne: https://riptutorial.com/fr/pandas/topic/6425/gotchas-de-pandas

Chapitre 14: Graphes et Visualisations

Examples

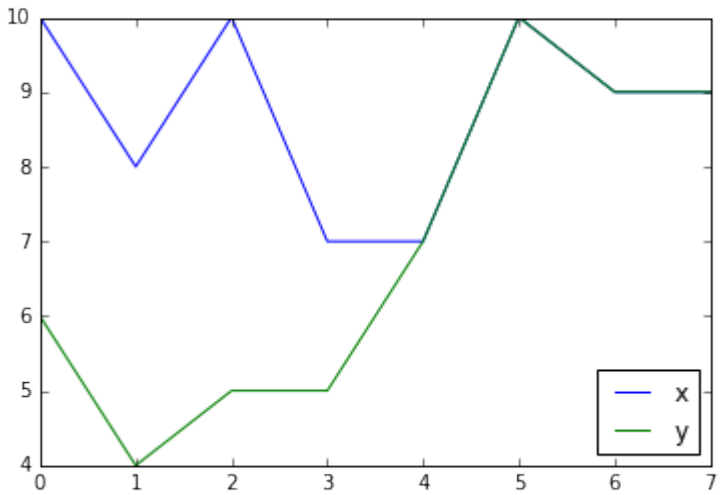
Graphiques de données de base

Pandas utilise plusieurs méthodes pour créer des graphiques des données dans le bloc de données. Il utilise du matplotlib à cette fin.

Les graphiques de base ont leurs enveloppes pour les objets DataFrame et Series:

Graphique linéaire

```
df = pd.DataFrame({'x': [10, 8, 10, 7, 7, 10, 9, 9],
                   'y': [6, 4, 5, 5, 7, 10, 9, 9]})
df.plot()
```



Vous pouvez appeler la même méthode pour un objet Series pour tracer un sous-ensemble du Data Frame:

```
df['x'].plot()
```

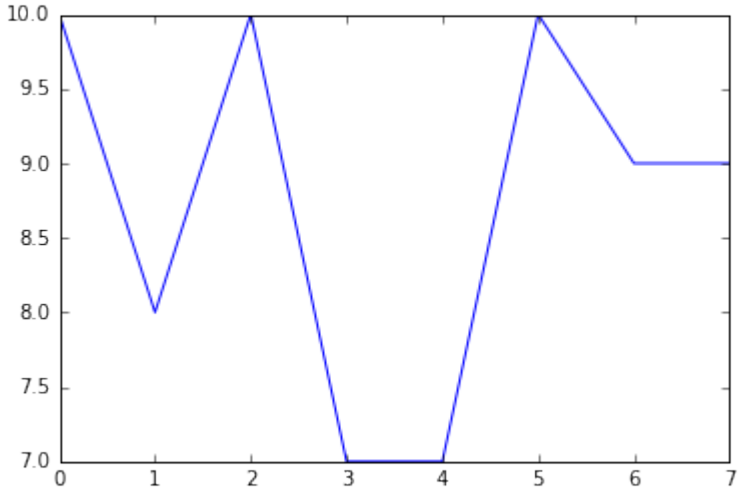
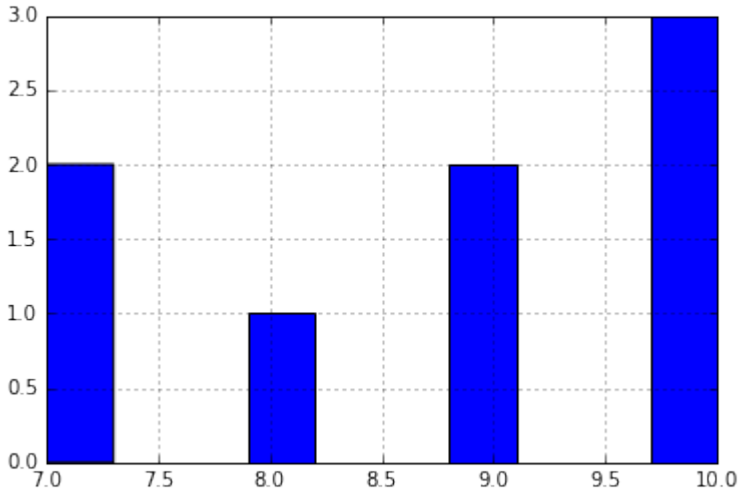


Diagramme à bandes

Si vous souhaitez explorer la distribution de vos données, vous pouvez utiliser la méthode `hist()` .

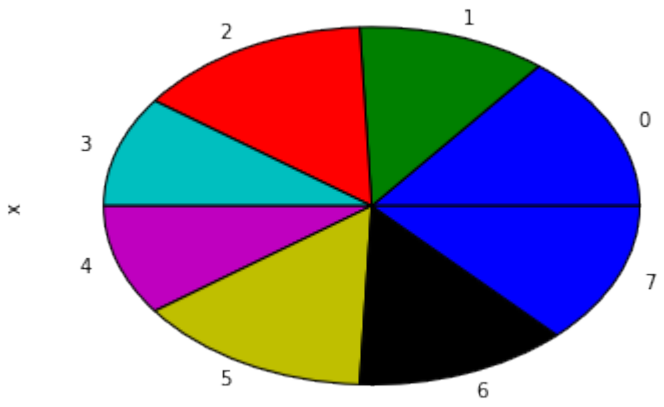
```
df['x'].hist()
```



Méthode générale pour tracer un tracé ()

Tous les graphiques possibles sont disponibles via la méthode de tracé. Le type de graphique est sélectionné par l'argument **kind** .

```
df['x'].plot(kind='pie')
```



Remarque Dans de nombreux environnements, le graphique à secteurs sortira un ovale. Pour en faire un cercle, utilisez ce qui suit:

```
from matplotlib import pyplot

pyplot.axis('equal')
df['x'].plot(kind='pie')
```

Styling l'intrigue

`plot()` peut prendre des arguments transmis à matplotlib pour donner un style à l'intrigue de différentes manières.

```
df.plot(style='o')  # plot as dots, not lines
df.plot(style='g--')  # plot as green dashed line
df.plot(style='o', markeredgecolor='white')  # plot as dots with white edge
```

Tracer sur un axe matplotlib existant

Par défaut, `plot()` crée un nouveau chiffre à chaque appel. Il est possible de tracer un axe existant en passant le paramètre `ax` .

```
plt.figure()  # create a new figure
ax = plt.subplot(121)  # create the left-side subplot
df1.plot(ax=ax)  # plot df1 on that subplot
ax = plt.subplot(122)  # create the right-side subplot
df2.plot(ax=ax)  # and plot df2 there
plt.show()  # show the plot
```

Lire Graphes et Visualisations en ligne: https://riptutorial.com/fr/pandas/topic/3839/graphes-et-visualisations

Chapitre 15: Indexation booléenne des dataframes

Introduction

Accès aux lignes d'un fichier de données à l'aide des objets d'indexeur `.ix` , `.loc` , `.iloc` et de la manière dont il se différencie de l'utilisation d'un masque booléen.

Examples

Accéder à un DataFrame avec un index booléen

Ce sera notre exemple de trame de données:

```
df = pd.DataFrame({"color": ['red', 'blue', 'red', 'blue']},
                  index=[True, False, True, False])
      color
True    red
False  blue
True    red
False  blue
```

Accéder avec `.loc`

```
df.loc[True]
     color
True   red
True   red
```

Accéder avec `.iloc`

```
df.iloc[True]
>> TypeError

df.iloc[1]
color    blue
dtype: object
```

> Il est important de noter que les anciennes versions de pandas ne faisaient pas de distinction entre les entrées booléennes et les entrées entières, donc `.iloc[True]` renverrait la même chose que `.iloc[1]`

Accéder avec `.ix`

```
df.ix[True]
     color
True   red
```

```
True   red

df.ix[1]
color    blue
dtype: object
```

Comme vous pouvez le voir, `.ix` a deux comportements. Ceci est une très mauvaise pratique en code et doit donc être évité. Veuillez utiliser `.iloc` ou `.loc` pour être plus explicite.

Application d'un masque booléen à un dataframe

Ce sera notre exemple de trame de données:

```
  color      name   size
0   red      rose    big
1  blue    violet    big
2   red     tulip  small
3  blue  harebell  small
```

En utilisant l' `__getitem__` magique `__getitem__` ou `[]` . En lui donnant une liste de True et False de la même longueur que le dataframe vous donnera:

```
df[[True, False, True, False]]
  color   name   size
0   red   rose    big
2   red  tulip  small
```

Masquage des données en fonction de la valeur de la colonne

Ce sera notre exemple de trame de données:

```
  color      name   size
0   red      rose    big
1  blue    violet  small
2   red     tulip  small
3  blue  harebell  small
```

En accédant à une seule colonne à partir d'un `pd.Series` de données, nous pouvons utiliser une simple comparaison `==` pour comparer chaque élément de la colonne à la variable donnée, produisant une `pd.Series` de True et False.

```
df['size'] == 'small'
0    False
1     True
2     True
3     True
Name: size, dtype: bool
```

Cette `pd.Series` est une extension d'un `np.array` qui est une extension d'une `list` simple. Ainsi, nous pouvons transmettre ceci à l' `__getitem__` ou `[]` comme dans l'exemple ci-dessus.

```
size_small_mask = df['size'] == 'small'
df[size_small_mask]
  color      name   size
1  blue    violet  small
2   red     tulip  small
3  blue  harebell  small
```

Masquage des données en fonction de la valeur d'index

Ce sera notre exemple de trame de données:

```
         color   size
name
rose       red    big
violet    blue  small
tulip      red  small
harebell  blue  small
```

Nous pouvons créer un masque basé sur les valeurs d'index, comme sur une valeur de colonne.

```
rose_mask = df.index == 'rose'
df[rose_mask]
     color size
name
rose   red  big
```

Mais faire cela est *presque* la même que

```
df.loc['rose']
color    red
size     big
Name: rose, dtype: object
```

La différence importante étant que, lorsque `.loc` ne rencontre qu'une ligne dans l'index correspondant, il retournera un `pd.Series` , s'il rencontre plus de lignes qui correspondent, il retournera un `pd.DataFrame` . Cela rend cette méthode plutôt instable.

Ce comportement peut être contrôlé en donnant à la `.loc` une liste d'une seule entrée. Cela l'obligera à retourner un bloc de données.

```
df.loc[['rose']]
         color   size
name
rose       red    big
```

Lire Indexation booléenne des dataframes en ligne:
https://riptutorial.com/fr/pandas/topic/9589/indexation-booleenne-des-dataframes

Chapitre 16: Indexation et sélection de données

Examples

Sélectionnez colonne par étiquette

```
# Create a sample DF
df = pd.DataFrame(np.random.randn(5, 3), columns=list('ABC'))

# Show DF
df
          A         B         C
0 -0.467542  0.469146 -0.861848
1 -0.823205 -0.167087 -0.759942
2 -1.508202  1.361894 -0.166701
3  0.394143 -0.287349 -0.978102
4 -0.160431  1.054736 -0.785250

# Select column using a single label, 'A'
df['A']
0   -0.467542
1   -0.823205
2   -1.508202
3    0.394143
4   -0.160431

# Select multiple columns using an array of labels, ['A', 'C']
df[['A', 'C']]
          A         C
0 -0.467542 -0.861848
1 -0.823205 -0.759942
2 -1.508202 -0.166701
3  0.394143 -0.978102
4 -0.160431 -0.785250
```

Détails supplémentaires sur: http://pandas.pydata.org/pandas-docs/version/0.18.0/indexing.html#selection-by-label

Sélectionner par position

La `iloc` méthode (abréviation de l' *emplacement entier)* permet de sélectionner les lignes d'une trame de données en fonction de leur indice de position. De cette façon, on peut découper des cadres de données comme on le fait avec le découpage des listes de Python.

```
df = pd.DataFrame([[11, 22], [33, 44], [55, 66]], index=list("abc"))

df
# Out:
#     0   1
# a  11  22
```

```
# b  33  44
# c  55  66

df.iloc[0]  # the 0th index (row)
# Out:
# 0    11
# 1    22
# Name: a, dtype: int64

df.iloc[1]  # the 1st index (row)
# Out:
# 0    33
# 1    44
# Name: b, dtype: int64

df.iloc[:2] # the first 2 rows
#     0   1
# a  11  22
# b  33  44

df[::-1]    # reverse order of rows
#     0   1
# c  55  66
# b  33  44
# a  11  22
```

L'emplacement des lignes peut être combiné avec l'emplacement des colonnes

```
df.iloc[:, 1]  # the 1st column
# Out[15]:
# a    22
# b    44
# c    66
# Name: 1, dtype: int64
```

Voir aussi: Sélection par position

Trancher avec des étiquettes

Lorsque vous utilisez des étiquettes, le début et la fin sont inclus dans les résultats.

```
import pandas as pd
import numpy as np
np.random.seed(5)
df = pd.DataFrame(np.random.randint(100, size=(5, 5)), columns = list("ABCDE"),
                  index = ["R" + str(i) for i in range(5)])

# Out:
#      A   B   C   D   E
# R0  99  78  61  16  73
# R1   8  62  27  30  80
# R2   7  76  15  53  80
# R3  27  44  77  75  65
# R4  47  30  84  86  18
```

Lignes `R0` à `R2` :

```
df.loc['R0':'R2']
# Out:
#      A   B   C   D   E
# R0   9  41  62   1  82
# R1  16  78   5  58   0
# R2  80   4  36  51  27
```

Notez que `loc` diffère d' `iloc` car `iloc` exclut l'index de fin

```
df.loc['R0':'R2'] # rows labelled R0, R1, R2
# Out:
#      A   B   C   D   E
# R0   9  41  62   1  82
# R1  16  78   5  58   0
# R2  80   4  36  51  27


# df.iloc[0:2] # rows indexed by 0, 1
#      A   B   C   D   E
# R0  99  78  61  16  73
# R1   8  62  27  30  80
```

Colonnes `C` à `E` :

```
df.loc[:, 'C':'E']
# Out:
#      C   D   E
# R0  62   1  82
# R1   5  58   0
# R2  36  51  27
# R3  68  38  83
# R4   7  30  62
```

Sélection mixte et sélection basée sur une étiquette

Trame de données:

```
import pandas as pd
import numpy as np
np.random.seed(5)
df = pd.DataFrame(np.random.randint(100, size=(5, 5)), columns = list("ABCDE"),
                  index = ["R" + str(i) for i in range(5)])


df
Out[12]:
     A   B   C   D   E
R0  99  78  61  16  73
R1   8  62  27  30  80
R2   7  76  15  53  80
R3  27  44  77  75  65
R4  47  30  84  86  18
```

Sélectionnez les lignes par position et les colonnes par libellé:

```
df.ix[1:3, 'C':'E']
Out[19]:
     C   D   E
R1   5  58   0
R2  36  51  27
```

Si l'index est entier, `.ix` utilisera des libellés plutôt que des positions:

```
df.index = np.arange(5, 10)

df
Out[22]:
    A   B   C   D   E
5   9  41  62   1  82
6  16  78   5  58   0
7  80   4  36  51  27
8  31   2  68  38  83
9  19  18   7  30  62

#same call returns an empty DataFrame because now the index is integer
df.ix[1:3, 'C':'E']
Out[24]:
Empty DataFrame
Columns: [C, D, E]
Index: []
```

Indexation booléenne

On peut sélectionner des lignes et des colonnes d'un dataframe en utilisant des tableaux booléens.

```
import pandas as pd
import numpy as np
np.random.seed(5)
df = pd.DataFrame(np.random.randint(100, size=(5, 5)), columns = list("ABCDE"),
                  index = ["R" + str(i) for i in range(5)])
print (df)
#      A   B   C   D   E
# R0  99  78  61  16  73
# R1   8  62  27  30  80
# R2   7  76  15  53  80
# R3  27  44  77  75  65
# R4  47  30  84  86  18
```

```
mask = df['A'] > 10
print (mask)
# R0     True
# R1    False
# R2    False
# R3     True
# R4     True
# Name: A, dtype: bool

print (df[mask])
#      A   B   C   D   E
# R0  99  78  61  16  73
# R3  27  44  77  75  65
```

```
# R4  47  30  84  86  18

print (df.ix[mask, 'C'])
# R0    61
# R3    77
# R4    84
# Name: C, dtype: int32

print(df.ix[mask, ['C', 'D']])
#       C   D
# R0  61  16
# R3  77  75
# R4  84  86
```

Plus dans la documentation des pandas .

Filtrage des colonnes (en sélectionnant "intéressant", en supprimant des données inutiles, en utilisant RegEx, etc.)

générer un échantillon DF

```
In [39]: df = pd.DataFrame(np.random.randint(0, 10, size=(5, 6)),
columns=['a10','a20','a25','b','c','d'])

In [40]: df
Out[40]:
   a10  a20  a25  b  c  d
0    2    3    7  5  4  7
1    3    1    5  7  2  6
2    7    4    9  0  8  7
3    5    8    8  9  6  8
4    8    1    0  4  4  9
```

affiche les colonnes contenant la lettre 'a'

```
In [41]: df.filter(like='a')
Out[41]:
   a10  a20  a25
0    2    3    7
1    3    1    5
2    7    4    9
3    5    8    8
4    8    1    0
```

affiche les colonnes à l'aide du filtre RegEx `(b|c|d)` - `b` ou `c` ou `d` :

```
In [42]: df.filter(regex='(b|c|d)')
```

```
Out[42]:
   b  c  d
0  5  4  7
1  7  2  6
2  0  8  7
3  9  6  8
4  4  4  9
```

afficher toutes les colonnes sauf celles commençant par `a` (en d'autres termes, supprimer / supprimer toutes les colonnes satisfaisant à RegEx donné)

```
In [43]: df.ix[:, ~df.columns.str.contains('^a')]
Out[43]:
   b  c  d
0  5  4  7
1  7  2  6
2  0  8  7
3  9  6  8
4  4  4  9
```

Filtrage / sélection de lignes en utilisant la méthode `.query ()`

```
import pandas as pd
```

générer des DF aléatoires

```
df = pd.DataFrame(np.random.randint(0,10,size=(10, 3)), columns=list('ABC'))

In [16]: print(df)
   A  B  C
0  4  1  4
1  0  2  0
2  7  8  8
3  2  1  9
4  7  3  8
5  4  0  7
6  1  5  5
7  6  7  8
8  6  7  3
9  6  4  5
```

sélectionnez les lignes où les valeurs de la colonne `A > 2` et les valeurs de la colonne `B < 5`

```
In [18]: df.query('A > 2 and B < 5')
Out[18]:
   A  B  C
0  4  1  4
4  7  3  8
5  4  0  7
9  6  4  5
```

utiliser la méthode `.query()` avec des variables pour le filtrage

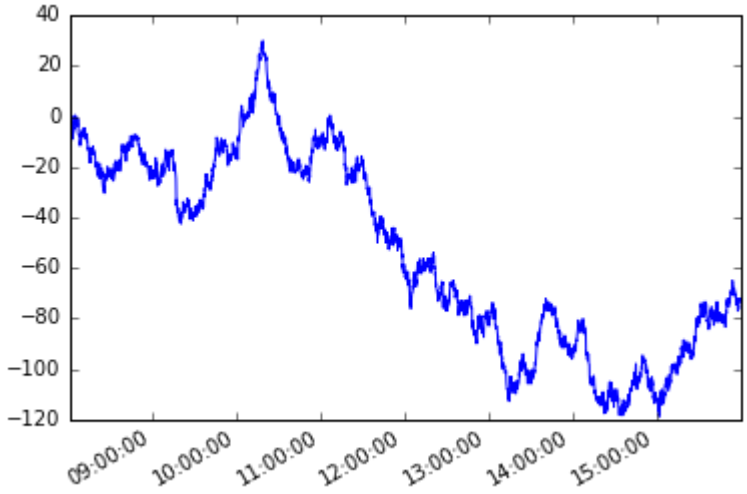
```
In [23]: B_filter = [1,7]

In [24]: df.query('B == @B_filter')
Out[24]:
   A  B  C
0  4  1  4
3  2  1  9
7  6  7  8
8  6  7  3

In [25]: df.query('@B_filter in B')
Out[25]:
   A  B  C
0  4  1  4
```

Tranchage dépendant du chemin

Il peut être nécessaire de parcourir les éléments d'une série ou les lignes d'un dataframe de manière à ce que l'élément suivant ou la ligne suivante dépende de l'élément ou de la ligne précédemment sélectionnés. Ceci s'appelle la dépendance de chemin.

Considérons la série chronologique suivant `s` avec une fréquence irrégulière.

```
#starting python community conventions
import numpy    as np
import pandas   as pd

# n is number of observations
n = 5000

day = pd.to_datetime(['2013-02-06'])
# irregular seconds spanning 28800 seconds (8 hours)
seconds = np.random.rand(n) * 28800 * pd.Timedelta(1, 's')
# start at 8 am
start = pd.offsets.Hour(8)
# irregular timeseries
tidx = day + start + seconds
tidx = tidx.sort_values()

s = pd.Series(np.random.randn(n), tidx, name='A').cumsum()
s.plot();
```



Supposons une condition dépendant du chemin. En commençant par le premier membre de la série, je veux saisir chaque élément suivant de sorte que la différence absolue entre cet élément et l'élément actuel soit supérieure ou égale à `x` .

Nous allons résoudre ce problème en utilisant des générateurs de python.

Fonction générateur

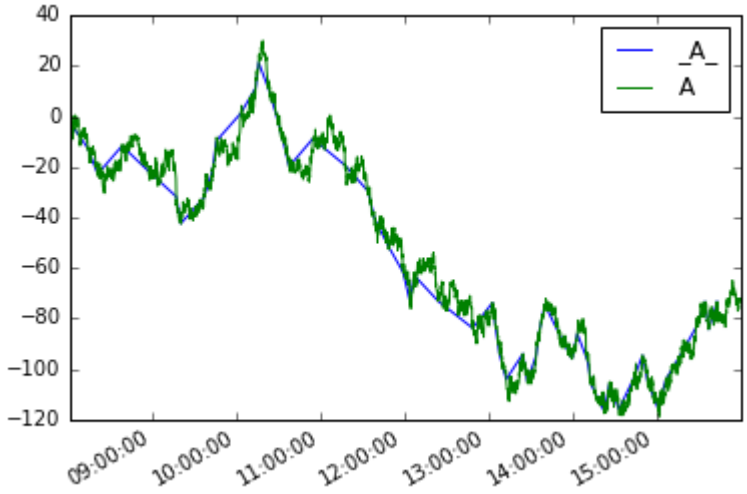
```
def mover(s, move_size=10):
    """Given a reference, find next value with
    an absolute difference >= move_size"""
    ref = None
    for i, v in s.iteritems():
        if ref is None or (abs(ref - v) >= move_size):
            yield i, v
            ref = v
```

Ensuite, nous pouvons définir une nouvelle série de `moves` comme ça

```
moves = pd.Series({i:v for i, v in mover(s, move_size=10)},
                  name='_{}_'.format(s.name))
```

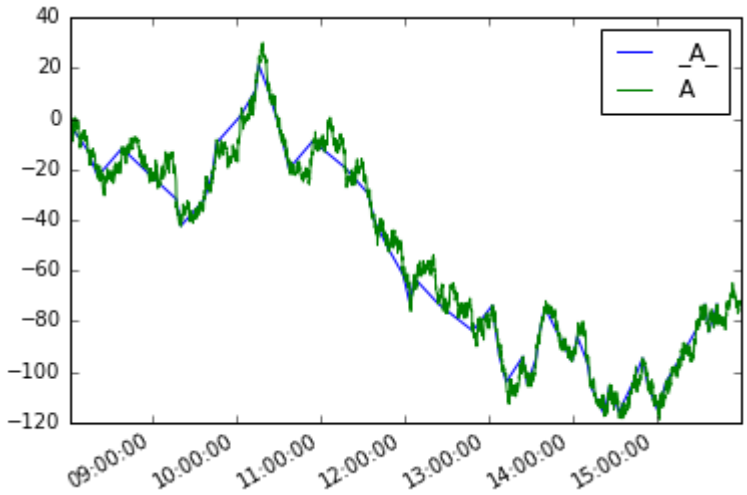
Les tracer tous les deux

```
moves.plot(legend=True)
s.plot(legend=True)
```



L'analogique pour les dataframes serait:

```
def mover_df(df, col, move_size=2):
    ref = None
    for i, row in df.iterrows():
        if ref is None or (abs(ref - row.loc[col]) >= move_size):
            yield row
            ref = row.loc[col]

df = s.to_frame()
moves_df = pd.concat(mover_df(df, 'A', 10), axis=1).T

moves_df.A.plot(label='_A_', legend=True)
df.A.plot(legend=True)
```



Récupère les premières / dernières n lignes d'un dataframe

Pour voir le premier ou le dernier enregistrement d'un dataframe, vous pouvez utiliser les méthodes `head` et `tail`

Pour renvoyer les n premières lignes, utilisez `DataFrame.head([n])`

```
df.head(n)
```

Pour retourner les n dernières lignes, utilisez `DataFrame.tail([n])`

```
df.tail(n)
```

Sans l'argument n, ces fonctions renvoient 5 lignes.

Notez que la notation de tranche pour `head` / `tail` serait:

```
df[:10]  # same as df.head(10)
df[-10:] # same as df.tail(10)
```

Sélectionnez des lignes distinctes sur l'ensemble des données

Laisser

```
df = pd.DataFrame({'col_1':['A','B','A','B','C'], 'col_2':[3,4,3,5,6]})
df
# Output:
#   col_1  col_2
# 0     A      3
# 1     B      4
# 2     A      3
# 3     B      5
# 4     C      6
```

Pour obtenir les valeurs distinctes dans `col_1` vous pouvez utiliser `Series.unique()`

```
df['col_1'].unique()
# Output:
# array(['A', 'B', 'C'], dtype=object)
```

Mais Series.unique () ne fonctionne que pour une seule colonne.

Pour simuler la *sélection unique col_1, col_2* de SQL, vous pouvez utiliser `DataFrame.drop_duplicates()` :

```
df.drop_duplicates()
#   col_1  col_2
# 0     A      3
# 1     B      4
# 3     B      5
# 4     C      6
```

Cela vous donnera toutes les lignes uniques dans le dataframe. Donc si

```
df = pd.DataFrame({'col_1':['A','B','A','B','C'], 'col_2':[3,4,3,5,6],
'col_3':[0,0.1,0.2,0.3,0.4]})
df
# Output:
#   col_1  col_2  col_3
# 0     A      3    0.0
# 1     B      4    0.1
```

```
# 2     A      3    0.2
# 3     B      5    0.3
# 4     C      6    0.4

df.drop_duplicates()
#   col_1  col_2  col_3
# 0     A      3    0.0
# 1     B      4    0.1
# 2     A      3    0.2
# 3     B      5    0.3
# 4     C      6    0.4
```

Pour spécifier les colonnes à prendre en compte lors de la sélection d'enregistrements uniques, transmettez-les comme arguments

```
df = pd.DataFrame({'col_1':['A','B','A','B','C'], 'col_2':[3,4,3,5,6],
'col_3':[0,0.1,0.2,0.3,0.4]})
df.drop_duplicates(['col_1','col_2'])
# Output:
#   col_1  col_2  col_3
# 0     A      3    0.0
# 1     B      4    0.1
# 3     B      5    0.3
# 4     C      6    0.4

# skip last column
# df.drop_duplicates(['col_1','col_2'])[['col_1','col_2']]
#   col_1  col_2
# 0     A      3
# 1     B      4
# 3     B      5
# 4     C      6
```

Source: Comment «sélectionner distinct» sur plusieurs colonnes de trames de données dans les pandas? .

Filtrer les lignes avec les données manquantes (NaN, None, NaT)

Si vous avez un dataframe avec des données manquantes (`NaN` , `pd.NaT` , `None`), vous pouvez filtrer les lignes incomplètes

```
df = pd.DataFrame([[0,1,2,3],
                   [None,5,None,pd.NaT],
                   [8,None,10,None],
                   [11,12,13,pd.NaT]],columns=list('ABCD'))
df
# Output:
#      A   B   C     D
# 0    0   1   2     3
# 1  NaN   5 NaN   NaT
# 2    8 NaN  10  None
# 3   11  12  13   NaT
```

`DataFrame.dropna` supprime toutes les lignes contenant au moins un champ avec des données manquantes

```
df.dropna()
# Output:
#    A  B  C  D
# 0  0  1  2  3
```

Pour simplement supprimer les lignes pour lesquelles il manque des données à des colonnes spécifiées, utilisez le `subset`

```
df.dropna(subset=['C'])
# Output:
#     A   B   C     D
# 0   0   1   2     3
# 2   8 NaN  10  None
# 3  11  12  13   NaT
```

Utilisez l'option `inplace = True` pour le remplacement sur place avec le cadre filtré.

Lire Indexation et sélection de données en ligne:
https://riptutorial.com/fr/pandas/topic/1751/indexation-et-selection-de-donnees

Chapitre 17: IO pour Google BigQuery

Examples

Lecture des données de BigQuery avec les informations d'identification du compte utilisateur

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

Pour exécuter une requête dans BigQuery, vous devez avoir votre propre projet BigQuery. Nous pouvons demander des exemples de données publiques:

```
In [2]: data = pd.read_gbq('''SELECT title, id, num_characters
   ...:                       FROM [publicdata:samples.wikipedia]
   ...:                       LIMIT 5'''
   ...:                     , project_id='<your-project-id>')
```

Cela va imprimer:

```
Your browser has been opened to visit:

    https://accounts.google.com/o/oauth2/v2/auth...[looong url cutted]

If your browser is on a different machine then exit and re-run this
application with the command-line parameter

  --noauth_local_webserver
```

Si vous opérez depuis un ordinateur local, le navigateur apparaîtra. Après avoir accordé des privilèges, les pandas continueront avec la sortie:

```
Authentication successful.
Requesting query... ok.
Query running...
Query done.
Processed: 13.8 Gb

Retrieving results...
Got 5 rows.

Total time taken 1.5 s.
Finished at 2016-08-23 11:26:03.
```

Résultat:

```
In [3]: data
Out[3]:
               title       id  num_characters
0       Fusidic acid   935328            1112
1     Clark Air Base   426241            8257
```

```
2  Watergate scandal    52382           25790
3               2005    35984           75813
4               .BLP  2664340            1659
```

Comme effet secondaire, les pandas créeront le fichier json `bigquery_credentials.dat` qui vous permettra d'exécuter d'autres requêtes sans avoir à accorder de privilèges:

```
In [9]: pd.read_gbq('SELECT count(1) cnt FROM [publicdata:samples.wikipedia]'
                    , project_id='<your-project-id>')
Requesting query... ok.
[rest of output cutted]

Out[9]:
         cnt
0  313797035
```

Lecture des données de BigQuery avec les informations d'identification du compte de service

Si vous avez créé un compte de service et que vous avez un fichier json de clé privée, vous pouvez utiliser ce fichier pour vous authentifier avec des pandas

```
In [5]: pd.read_gbq('''SELECT corpus, sum(word_count) words
                       FROM [bigquery-public-data:samples.shakespeare]
                       GROUP BY corpus
                       ORDER BY words desc
                       LIMIT 5'''
                    , project_id='<your-project-id>'
                    , private_key='<private key json contents or file path>')
Requesting query... ok.
[rest of output cutted]

Out[5]:
           corpus  words
0          hamlet  32446
1  kingrichardiii  31868
2      coriolanus  29535
3       cymbeline  29231
4    2kinghenryiv  28241
```

Lire IO pour Google BigQuery en ligne: https://riptutorial.com/fr/pandas/topic/5610/io-pour-google-bigquery

Chapitre 18: JSON

Examples

Lire JSON

peut soit transmettre une chaîne de json, soit un chemin de fichier à un fichier avec json valide

```
In [99]: pd.read_json('[{"A": 1, "B": 2}, {"A": 3, "B": 4}]')
Out[99]:
   A  B
0  1  2
1  3  4
```

Sinon, conservez de la mémoire:

```
with open('test.json') as f:
    data = pd.DataFrame(json.loads(line) for line in f)
```

Dataframe dans JSON imbriqué comme dans les fichiers flare.js utilisés dans D3.js

```
def to_flare_json(df, filename):
    """Convert dataframe into nested JSON as in flare files used for D3.js"""
    flare = dict()
    d = {"name":"flare", "children": []}

    for index, row in df.iterrows():
        parent = row[0]
        child = row[1]
        child_size = row[2]

        # Make a list of keys
        key_list = []
        for item in d['children']:
            key_list.append(item['name'])

        #if 'parent' is NOT a key in flare.JSON, append it
        if not parent in key_list:
            d['children'].append({"name": parent, "children":[{"value": child_size, "name":
child}]})
        # if parent IS a key in flare.json, add a new child to it
        else:
            d['children'][key_list.index(parent)]['children'].append({"value": child_size,
"name": child})
```

```
    flare = d
    # export the final result to a json file
    with open(filename +'.json', 'w') as outfile:
        json.dump(flare, outfile, indent=4)
    return ("Done")
```

Lire JSON à partir du fichier

Contenu de file.json (un objet JSON par ligne):

```
{"A": 1, "B": 2}
{"A": 3, "B": 4}
```

Comment lire directement depuis un fichier local:

```
pd.read_json('file.json', lines=True)
# Output:
#    A  B
# 0  1  2
# 1  3  4
```

Lire JSON en ligne: https://riptutorial.com/fr/pandas/topic/4752/json

Chapitre 19: Lecture de fichiers dans des pandas DataFrame

Examples

Lire la table dans DataFrame

Fichier de table avec en-tête, pied de page, noms de ligne et colonne d'index:

fichier: table.txt

```
This is a header that discusses the table file
to show space in a generic table file

index  name     occupation
1      Alice    Salesman
2      Bob      Engineer
3      Charlie  Janitor

This is a footer because your boss does not understand data files
```

code:

```
import pandas as pd
# index_col=0 tells pandas that column 0 is the index and not data
pd.read_table('table.txt', delim_whitespace=True, skiprows=3, skipfooter=2, index_col=0)
```

sortie:

```
          name occupation
index
1        Alice   Salesman
2          Bob   Engineer
3      Charlie    Janitor
```

Fichier de table sans noms de lignes ou index:

fichier: table.txt

```
Alice    Salesman
Bob      Engineer
Charlie  Janitor
```

code:

```
import pandas as pd
pd.read_table('table.txt', delim_whitespace=True, names=['name','occupation'])
```

sortie:

```
      name occupation
0    Alice   Salesman
1      Bob   Engineer
2  Charlie    Janitor
```

Toutes les options peuvent être trouvées dans la documentation des pandas ici

Lire un fichier CSV

Données avec en-tête, séparées par des points-virgules au lieu de virgules

fichier: table.csv

```
index;name;occupation
1;Alice;Saleswoman
2;Bob;Engineer
3;Charlie;Janitor
```

code:

```
import pandas as pd
pd.read_csv('table.csv', sep=';', index_col=0)
```

sortie :

```
          name occupation
index
1        Alice   Salesman
2          Bob   Engineer
3      Charlie    Janitor
```

Table sans noms de lignes ou index et virgules comme séparateurs

fichier: table.csv

```
Alice,Saleswoman
Bob,Engineer
Charlie,Janitor
```

code:

```
import pandas as pd
pd.read_csv('table.csv', names=['name','occupation'])
```

sortie:

```
      name occupation
0    Alice   Salesman
1      Bob   Engineer
2  Charlie    Janitor
```

des précisions supplémentaires peuvent être trouvées dans la page de documentation de `read_csv`

Recueillez les données de la feuille de calcul google dans les données pandas

Parfois, nous devons collecter des données à partir de feuilles de calcul google. Nous pouvons utiliser les bibliothèques **gspread** et **oauth2client** pour collecter des données à partir de feuilles de calcul google. Voici un exemple pour collecter des données:

Code:

```
from __future__ import print_function
import gspread
from oauth2client.client import SignedJwtAssertionCredentials
import pandas as pd
import json

scope = ['https://spreadsheets.google.com/feeds']

credentials = ServiceAccountCredentials.from_json_keyfile_name('your-authorization-file.json',
scope)

gc = gspread.authorize(credentials)

work_sheet = gc.open_by_key("spreadsheet-key-here")
sheet = work_sheet.sheet1
data = pd.DataFrame(sheet.get_all_records())

print(data.head())
```

Lire Lecture de fichiers dans des pandas DataFrame en ligne:
https://riptutorial.com/fr/pandas/topic/1988/lecture-de-fichiers-dans-des-pandas-dataframe

Chapitre 20: Lire MySQL sur DataFrame

Examples

Utiliser sqlalchemy et PyMySQL

```
from sqlalchemy import create_engine

cnx = create_engine('mysql+pymysql://username:password@server:3306/database').connect()
sql = 'select * from mytable'
df = pd.read_sql(sql, cnx)
```

Pour lire mysql sur dataframe, en cas de grande quantité de données

Pour récupérer des données volumineuses, nous pouvons utiliser des générateurs dans les pandas et charger des données en morceaux.

```
import pandas as pd
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL


# sqlalchemy engine
engine = create_engine(URL(
    drivername="mysql"
    username="user",
    password="password"
    host="host"
    database="database"
))

conn = engine.connect()

generator_df = pd.read_sql(sql=query,  # mysql query
                           con=conn,
                           chunksize=chunksize)  # size you want to fetch each time

for dataframe in generator_df:
    for row in dataframe:
        pass  # whatever you want to do
```

Lire Lire MySQL sur DataFrame en ligne: https://riptutorial.com/fr/pandas/topic/8809/lire-mysql-sur-dataframe

Chapitre 21: Lire SQL Server vers Dataframe

Examples

Utiliser pyodbc

```
import pandas.io.sql
import pyodbc
import pandas as pd
```

Spécifiez les paramètres

```
# Parameters
server = 'server_name'
db = 'database_name'
UID = 'user_id'
```

Créer la connexion

```
# Create the connection
conn = pyodbc.connect('DRIVER={SQL Server};SERVER=' + server + ';DATABASE=' + db + '; UID = '
+ UID + '; PWD = ' + UID + 'Trusted_Connection=yes')
```

Requête en pandas dataframe

```
# Query into dataframe
df= pandas.io.sql.read_sql('sql_query_string', conn)
```

Utiliser pyodbc avec boucle de connexion

```
import os, time
import pyodbc
import pandas.io.sql as pdsql

def todf(dsn='yourdsn', uid=None, pwd=None, query=None, params=None):
    ''' if `query` is not an actual query but rather a path to a text file
        containing a query, read it in instead '''
    if query.endswith('.sql') and os.path.exists(query):
        with open(query,'r') as fin:
            query = fin.read()

    connstr = "DSN={};UID={};PWD={}".format(dsn,uid,pwd)
    connected = False
    while not connected:
        try:
            with pyodbc.connect(connstr,autocommit=True) as con:
                cur = con.cursor()
                if params is not None: df = pdsql.read_sql(query, con,
                                                           params=params)
                else: df = pdsql.read_sql(query, con)
                cur.close()
```

```
            break
        except pyodbc.OperationalError:
            time.sleep(60) # one minute could be changed
    return df
```

Lire Lire SQL Server vers Dataframe en ligne: https://riptutorial.com/fr/pandas/topic/2176/lire-sql-server-vers-dataframe

Chapitre 22: Manipulation de cordes

Examples

Expressions régulières

```
# Extract strings with a specific regex
df= df['col_name'].str.extract[r'[Aa-Zz]']

# Replace strings within a regex
df['col_name'].str.replace('Replace this', 'With this')
```

Pour plus d'informations sur la manière de faire correspondre les chaînes à l'aide de l'expression rationnelle, voir Mise en route avec les expressions régulières .

Ficelle

Les chaînes d'une série peuvent être découpées en utilisant la méthode `.str.slice()` , ou plus facilement, entre parenthèses (`.str[]`).

```
In [1]: ser = pd.Series(['Lorem ipsum', 'dolor sit amet', 'consectetur adipiscing elit'])
In [2]: ser
Out[2]:
0                   Lorem ipsum
1                 dolor sit amet
2    consectetur adipiscing elit
dtype: object
```

Obtenez le premier caractère de chaque chaîne:

```
In [3]: ser.str[0]
Out[3]:
0    L
1    d
2    c
dtype: object
```

Obtenez les trois premiers caractères de chaque chaîne:

```
In [4]: ser.str[:3]
Out[4]:
0    Lor
1    dol
2    con
dtype: object
```

Récupère le dernier caractère de chaque chaîne:

```
In [5]: ser.str[-1]
```

```
Out[5]:
0    m
1    t
2    t
dtype: object
```

Obtenez les trois derniers caractères de chaque chaîne:

```
In [6]: ser.str[-3:]
Out[6]:
0    sum
1    met
2    lit
dtype: object
```

Obtenez tous les autres caractères des 10 premiers caractères:

```
In [7]: ser.str[:10:2]
Out[7]:
0    Lrmis
1    dlrst
2    cnett
dtype: object
```

Pandas se comporte de la même manière que Python lors de la manipulation de tranches et d'index. Par exemple, si un index est en dehors de la plage, Python génère une erreur:

```
In [8]:'Lorem ipsum'[12]
# IndexError: string index out of range
```

Cependant, si une tranche est en dehors de la plage, une chaîne vide est renvoyée:

```
In [9]: 'Lorem ipsum'[12:15]
Out[9]: ''
```

Pandas renvoie NaN lorsqu'un index est hors limites:

```
In [10]: ser.str[12]
Out[10]:
0    NaN
1      e
2      a
dtype: object
```

Et renvoie une chaîne vide si une tranche est hors limites:

```
In [11]: ser.str[12:15]
Out[11]:
0
1     et
2    adi
dtype: object
```

Vérification du contenu d'une chaîne

`str.contains()` méthode `str.contains()` peut être utilisée pour vérifier si un motif se produit dans chaque chaîne d'une série. `str.startswith()` et `str.endswith()` peuvent également être utilisées comme versions plus spécialisées.

```
In [1]: animals = pd.Series(['cat', 'dog', 'bear', 'cow', 'bird', 'owl', 'rabbit', 'snake'])
```

Vérifiez si les chaînes contiennent la lettre 'a':

```
In [2]: animals.str.contains('a')
Out[2]:
0      True
1     False
2      True
3     False
4     False
5     False
6      True
7      True
8      True
dtype: bool
```

Ceci peut être utilisé comme un index booléen pour ne renvoyer que les animaux contenant la lettre 'a':

```
In [3]: animals[animals.str.contains('a')]
Out[3]:
0       cat
2      bear
6    rabbit
7     snake
dtype: object
```

`str.startswith` méthodes `str.startswith` et `str.endswith` fonctionnent de manière similaire, mais elles acceptent également les tuples comme entrées.

```
In [4]: animals[animals.str.startswith(('b', 'c'))]
# Returns animals starting with 'b' or 'c'
Out[4]:
0     cat
2    bear
3     cow
4    bird
dtype: object
```

Capitalisation de chaînes

```
In [1]: ser = pd.Series(['lORem ipSuM', 'Dolor sit amet', 'Consectetur Adipiscing Elit'])
```

Convertir tout en majuscule:

```
In [2]: ser.str.upper()
Out[2]:
0                    LOREM IPSUM
1                 DOLOR SIT AMET
2    CONSECTETUR ADIPISCING ELIT
dtype: object
```

Tout en minuscule:

```
In [3]: ser.str.lower()
Out[3]:
0                    lorem ipsum
1                 dolor sit amet
2    consectetur adipiscing elit
dtype: object
```

Capitaliser le premier caractère et minuscule le reste:

```
In [4]: ser.str.capitalize()
Out[4]:
0                    Lorem ipsum
1                 Dolor sit amet
2    Consectetur adipiscing elit
dtype: object
```

Convertissez chaque chaîne en une titlecase (mettez en majuscule le premier caractère de chaque mot dans chaque chaîne, minuscule le reste):

```
In [5]: ser.str.title()
Out[5]:
0                    Lorem Ipsum
1                 Dolor Sit Amet
2    Consectetur Adipiscing Elit
dtype: object
```

Permuter les cas (convertir les minuscules en majuscules et vice versa):

```
In [6]: ser.str.swapcase()
Out[6]:
0                    LorEM IPsUm
1                 dOLOR SIT AMET
2    cONSECTETUR aDIPISCING eLIT
dtype: object
```

Outre ces méthodes qui modifient la capitalisation, plusieurs méthodes peuvent être utilisées pour vérifier la capitalisation des chaînes.

```
In [7]: ser = pd.Series(['LOREM IPSUM', 'dolor sit amet', 'Consectetur Adipiscing Elit'])
```

Vérifiez si tout est en minuscule:

```
In [8]: ser.str.islower()
Out[8]:
```

```
0    False
1     True
2    False
dtype: bool
```

Est-ce tout en majuscule:

```
In [9]: ser.str.isupper()
Out[9]:
0     True
1    False
2    False
dtype: bool
```

Est-ce une chaîne titlecased:

```
In [10]: ser.str.istitle()
Out[10]:
0    False
1    False
2     True
dtype: bool
```

Lire Manipulation de cordes en ligne: https://riptutorial.com/fr/pandas/topic/2372/manipulation-de-cordes

Chapitre 23: Manipulation simple de DataFrames

Examples

Supprimer une colonne dans un DataFrame

Il existe plusieurs façons de supprimer une colonne dans un DataFrame.

```
import numpy as np
import pandas as pd

np.random.seed(0)

pd.DataFrame(np.random.randn(5, 6), columns=list('ABCDEF'))

print(df)
# Output:
#           A         B         C         D         E         F
# 0 -0.895467  0.386902 -0.510805 -1.180632 -0.028182  0.428332
# 1  0.066517  0.302472 -0.634322 -0.362741 -0.672460 -0.359553
# 2 -0.813146 -1.726283  0.177426 -0.401781 -1.630198  0.462782
# 3 -0.907298  0.051945  0.729091  0.128983  1.139401 -1.234826
# 4  0.402342 -0.684810 -0.870797 -0.578850 -0.311553  0.056165
```

1) Utiliser `del`

```
del df['C']

print(df)
# Output:
#           A         B         D         E         F
# 0 -0.895467  0.386902 -1.180632 -0.028182  0.428332
# 1  0.066517  0.302472 -0.362741 -0.672460 -0.359553
# 2 -0.813146 -1.726283 -0.401781 -1.630198  0.462782
# 3 -0.907298  0.051945  0.128983  1.139401 -1.234826
# 4  0.402342 -0.684810 -0.578850 -0.311553  0.056165
```

2) Utiliser `drop`

```
df.drop(['B', 'E'], axis='columns', inplace=True)
# or df = df.drop(['B', 'E'], axis=1) without the option inplace=True

print(df)
# Output:
#           A         D         F
# 0 -0.895467 -1.180632  0.428332
# 1  0.066517 -0.362741 -0.359553
# 2 -0.813146 -0.401781  0.462782
# 3 -0.907298  0.128983 -1.234826
# 4  0.402342 -0.578850  0.056165
```

3) Utilisation de `drop` avec les numéros de colonne

Pour utiliser des nombres entiers de colonne au lieu de noms (rappelez-vous que les index de colonne commencent à zéro):

```
df.drop(df.columns[[0, 2]], axis='columns')

print(df)
# Output:
#           D
# 0 -1.180632
# 1 -0.362741
# 2 -0.401781
# 3  0.128983
# 4 -0.578850
```

Renommer une colonne

```
df = pd.DataFrame({'old_name_1': [1, 2, 3], 'old_name_2': [5, 6, 7]})

print(df)
# Output:
#    old_name_1  old_name_2
# 0           1           5
# 1           2           6
# 2           3           7
```

Pour renommer une ou plusieurs colonnes, transmettez les anciens noms et les nouveaux noms en tant que dictionnaire:

```
df.rename(columns={'old_name_1': 'new_name_1', 'old_name_2': 'new_name_2'}, inplace=True)
print(df)
# Output:
#   new_name_1  new_name_2
# 0           1           5
# 1           2           6
# 2           3           7
```

Ou une fonction:

```
df.rename(columns=lambda x: x.replace('old_', '_new'), inplace=True)
print(df)
# Output:
#   new_name_1  new_name_2
# 0           1           5
# 1           2           6
# 2           3           7
```

Vous pouvez également définir `df.columns` comme liste des nouveaux noms:

```
df.columns = ['new_name_1','new_name_2']
print(df)
# Output:
#   new_name_1  new_name_2
```

```
# 0           1           5
# 1           2           6
# 2           3           7
```

Plus de détails peuvent être trouvés ici .

Ajouter une nouvelle colonne

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

print(df)
# Output:
#    A  B
# 0  1  4
# 1  2  5
# 2  3  6
```

Directement attribuer

```
df['C'] = [7, 8, 9]

print(df)
# Output:
#    A  B  C
# 0  1  4  7
# 1  2  5  8
# 2  3  6  9
```

Ajouter une colonne constante

```
df['C'] = 1

print(df)

# Output:
#    A  B  C
# 0  1  4  1
# 1  2  5  1
# 2  3  6  1
```

Colonne comme expression dans les autres colonnes

```
df['C'] = df['A'] + df['B']

# print(df)
# Output:
#    A  B  C
# 0  1  4  5
# 1  2  5  7
# 2  3  6  9

df['C'] = df['A']**df['B']
```

```
print(df)
# Output:
#    A  B    C
# 0  1  4    1
# 1  2  5   32
# 2  3  6  729
```

Les opérations sont calculées par composants, donc si nous avions des colonnes comme des listes

```
a = [1, 2, 3]
b = [4, 5, 6]
```

la colonne dans la dernière expression serait obtenue comme

```
c = [x**y for (x,y) in zip(a,b)]

print(c)
# Output:
# [1, 32, 729]
```

Créez-le à la volée

```
df_means = df.assign(D=[10, 20, 30]).mean()

print(df_means)
# Output:
# A     2.0
# B     5.0
# C     7.0
# D    20.0  # adds a new column D before taking the mean
# dtype: float64
```

ajouter plusieurs colonnes

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df[['A2','B2']] = np.square(df)

print(df)
# Output:
#    A  B  A2  B2
# 0  1  4   1  16
# 1  2  5   4  25
# 2  3  6   9  36
```

ajouter plusieurs colonnes à la volée

```
new_df = df.assign(A3=df.A*df.A2, B3=5*df.B)

print(new_df)
```

```
# Output:
#    A  B  A2  B2  A3  B3
# 0  1  4   1  16   1  20
# 1  2  5   4  25   8  25
# 2  3  6   9  36  27  30
```

Localiser et remplacer des données dans une colonne

```
import pandas as pd

df = pd.DataFrame({'gender': ["male", "female","female"],
                   'id': [1, 2, 3] })
>>> df
   gender  id
0    male   1
1  female   2
2  female   3
```

Pour encoder le mâle à 0 et le féminin à 1:

```
df.loc[df["gender"] == "male","gender"] = 0
df.loc[df["gender"] == "female","gender"] = 1

>>> df
       gender  id
    0       0   1
    1       1   2
    2       1   3
```

Ajout d'une nouvelle ligne à DataFrame

Étant donné un DataFrame:

```
s1 = pd.Series([1,2,3])
s2 = pd.Series(['a','b','c'])

df = pd.DataFrame([list(s1), list(s2)],  columns =  ["C1", "C2", "C3"])
print df
```

Sortie:

```
  C1 C2 C3
0  1  2  3
1  a  b  c
```

Permet d'ajouter une nouvelle ligne, `[10,11,12]` :

```
df = pd.DataFrame(np.array([[10,11,12]]), \
        columns=["C1", "C2", "C3"]).append(df, ignore_index=True)
print df
```

Sortie:

```
   C1  C2  C3
0  10  11  12
1   1   2   3
2   a   b   c
```

Supprimer / supprimer des lignes de DataFrame

générons d'abord un DataFrame:

```
df = pd.DataFrame(np.arange(10).reshape(5,2), columns=list('ab'))

print(df)
# Output:
#    a  b
# 0  0  1
# 1  2  3
# 2  4  5
# 3  6  7
# 4  8  9
```

déposer des lignes avec des index: `0` et `4` utilisant la méthode `drop([...], inplace=True)` :

```
df.drop([0,4], inplace=True)

print(df)
# Output
#    a  b
# 1  2  3
# 2  4  5
# 3  6  7
```

déposer des lignes avec des index: `0` et `4` utilisant la méthode `df = drop([...])` :

```
df = pd.DataFrame(np.arange(10).reshape(5,2), columns=list('ab'))

df = df.drop([0,4])

print(df)
# Output:
#    a  b
# 1  2  3
# 2  4  5
# 3  6  7
```

en utilisant la méthode de sélection négative:

```
df = pd.DataFrame(np.arange(10).reshape(5,2), columns=list('ab'))

df = df[~df.index.isin([0,4])]

print(df)
# Output:
#    a  b
# 1  2  3
# 2  4  5
```

```
# 3  6  7
```

Réorganiser les colonnes

```
# get a list of columns
cols = list(df)

# move the column to head of list using index, pop and insert
cols.insert(0, cols.pop(cols.index('listing')))

# use ix to reorder
df2 = df.ix[:, cols]
```

Lire Manipulation simple de DataFrames en ligne:
https://riptutorial.com/fr/pandas/topic/6694/manipulation-simple-de-dataframes

Chapitre 24: Meta: Guide de documentation

Remarques

Cette méta-post est similaire à la version de python http://stackoverflow.com/documentation/python/394/meta-documentation-guidelines#t=20160724005840635952 1 .

S'il vous plaît faire des suggestions d'édition et commenter celles-ci (au lieu de commentaires appropriés), afin que nous puissions étoffer / itérer sur ces suggestions :)

Examples

Affichage des extraits de code et sortie

Deux options populaires sont à utiliser:

notation ipython:

```
In [11]: df = pd.DataFrame([[1, 2], [3, 4]])

In [12]: df
Out[12]:
   0  1
0  1  2
1  3  4
```

Alternativement (c'est populaire dans la documentation de python) et de manière plus concise:

```
df.columns  # Out: RangeIndex(start=0, stop=2, step=1)

df[0]
# Out:
# 0    1
# 1    3
# Name: 0, dtype: int64

for col in df:
    print(col)
# prints:
# 0
# 1
```

Généralement, cela vaut mieux pour les plus petits exemples.

Remarque: la distinction entre sortie et impression. ipython le dit clairement (les impressions se produisent avant que la sortie ne soit renvoyée):

```
In [21]: [print(col) for col in df]
```

```
0
1
Out[21]: [None, None]
```

style

Utilisez la bibliothèque de pandas en tant que `pd` , cela peut être supposé (l'importation n'a pas besoin d'être dans chaque exemple)

```
import pandas as pd
```

PEP8!

- 4 indentation spatiale
- kwargs ne doit utiliser aucun espace `f(a=1)`
- Limite de 80 caractères (toute la ligne correspondant à l'extrait de code rendu doit être fortement préférée)

Prise en charge de la version Pandas

La plupart des exemples fonctionneront sur plusieurs versions, si vous utilisez une "nouvelle" fonctionnalité, vous devez indiquer quand elle a été introduite.

Exemple: `sort_values` .

imprimer des relevés

La plupart du temps, l'impression doit être évitée car cela peut être une distraction (la préférence doit être donnée à la sortie).
C'est:

```
a
# Out: 1
```

est toujours mieux que

```
print(a)
# prints: 1
```

Préférez le support de python 2 et 3:

```
print(x)    # yes! (works same in python 2 and 3)
print x     # no! (python 2 only)
print(x, y) # no! (works differently in python 2 and 3)
```

Lire Meta: Guide de documentation en ligne: https://riptutorial.com/fr/pandas/topic/3253/meta--guide-de-documentation

Chapitre 25: MultiIndex

Examples

Sélectionnez MultiIndex par niveau

Étant donné le DataFrame suivant:

```
In [11]: df = pd.DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])

In [12]: df.set_index(['A', 'B'], inplace=True)

In [13]: df
Out[13]:
                          C
A         B
 0.902764 -0.259656 -1.864541
-0.695893  0.308893  0.125199
 1.696989 -1.221131 -2.975839
-1.132069 -1.086189 -1.945467
 2.294835 -1.765507  1.567853
-1.788299  2.579029  0.792919
```

Obtenez les valeurs de `A` , par nom:

```
In [14]: df.index.get_level_values('A')
Out[14]:
Float64Index([0.902764041011, -0.69589264969,  1.69698924476, -1.13206872067,
              2.29483481146,   -1.788298829],
             dtype='float64', name='A')
```

Ou par nombre de niveau:

```
In [15]: df.index.get_level_values(level=0)
Out[15]:
Float64Index([0.902764041011, -0.69589264969,  1.69698924476, -1.13206872067,
              2.29483481146,   -1.788298829],
             dtype='float64', name='A')
```

Et pour une gamme spécifique:

```
In [16]: df.loc[(df.index.get_level_values('A') > 0.5) & (df.index.get_level_values('A') <
2.1)]
Out[16]:
                          C
A        B
0.902764 -0.259656 -1.864541
1.696989 -1.221131 -2.975839
```

La plage peut également inclure plusieurs colonnes:

```
In [17]: df.loc[(df.index.get_level_values('A') > 0.5) & (df.index.get_level_values('B') < 0)]
Out[17]:
                            C
A        B
0.902764 -0.259656 -1.864541
1.696989 -1.221131 -2.975839
2.294835 -1.765507  1.567853
```

Pour extraire une valeur spécifique, vous pouvez utiliser xs (section transversale):

```
In [18]: df.xs(key=0.9027639999999999)
Out[18]:
                  C
B
-0.259656 -1.864541

In [19]: df.xs(key=0.9027639999999999, drop_level=False)
Out[19]:
                            C
A        B
0.902764 -0.259656 -1.864541
```

Itérer sur DataFrame avec MultiIndex

Étant donné le DataFrame suivant:

```
In [11]: df = pd.DataFrame({'a':[1,1,1,2,2,3],'b':[4,4,5,5,6,7,],'c':[10,11,12,13,14,15]})

In [12]: df.set_index(['a','b'], inplace=True)

In [13]: df
Out[13]:
      c
a b
1 4  10
  4  11
  5  12
2 5  13
  6  14
3 7  15
```

Vous pouvez effectuer une itération par n'importe quel niveau du MultiIndex. Par exemple, `level=0` (vous pouvez également sélectionner le niveau par nom, par exemple `level='a'`):

```
In[21]: for idx, data in df.groupby(level=0):
            print('---')
            print(data)
---
      c
a b
1 4  10
  4  11
  5  12
---
      c
a b
```

```
2 5  13
  6  14
---
      c
a b
3 7  15
```

Vous pouvez également sélectionner les niveaux par nom, par exemple `level = 'b':

```
In[22]: for idx, data in df.groupby(level='b'):
            print('---')
            print(data)
---
      c
a b
1 4  10
  4  11
---
      c
a b
1 5  12
2 5  13
---
      c
a b
2 6  14
---
      c
a b
3 7  15
```

Définition et tri d'un MultiIndex

Cet exemple montre comment utiliser des données de colonne pour définir un `MultiIndex` dans un `pandas.DataFrame` .

```
In [1]: df = pd.DataFrame([['one', 'A', 100], ['two', 'A', 101], ['three', 'A', 102],
   ...:                    ['one', 'B', 103], ['two', 'B', 104], ['three', 'B', 105]],
   ...:                   columns=['c1', 'c2', 'c3'])


In [2]: df
Out[2]:
      c1 c2   c3
0    one  A  100
1    two  A  101
2  three  A  102
3    one  B  103
4    two  B  104
5  three  B  105



In [3]: df.set_index(['c1', 'c2'])
Out[3]:
           c3
c1    c2
```

```
one   A   100
two   A   101
three A   102
one   B   103
two   B   104
three B   105
```

Vous pouvez trier l'index juste après l'avoir défini:

```
In [4]: df.set_index(['c1', 'c2']).sort_index()
Out[4]:
           c3
c1    c2
one   A   100
      B   103
three A   102
      B   105
two   A   101
      B   104
```

Avoir un index trié entraînera des recherches légèrement plus efficaces au premier niveau:

```
In [5]: df_01 = df.set_index(['c1', 'c2'])

In [6]: %timeit df_01.loc['one']
1000 loops, best of 3: 607 µs per loop


In [7]: df_02 = df.set_index(['c1', 'c2']).sort_index()

In [8]: %timeit df_02.loc['one']
1000 loops, best of 3: 413 µs per loop
```

Une fois l'index défini, vous pouvez effectuer des recherches pour des enregistrements ou des groupes d'enregistrements spécifiques:

```
In [9]: df_indexed = df.set_index(['c1', 'c2']).sort_index()

In [10]: df_indexed.loc['one']
Out[10]:
     c3
c2
A   100
B   103


In [11]: df_indexed.loc['one', 'A']
Out[11]:
c3    100
Name: (one, A), dtype: int64


In [12]: df_indexed.xs((slice(None), 'A'))
Out[12]:
        c3
c1
one    100
```

```
three  102
two    101
```

Comment changer les colonnes MultiIndex en colonnes standard

Étant donné un DataFrame avec des colonnes MultiIndex

```
# build an example DataFrame
midx = pd.MultiIndex(levels=[['zero', 'one'], ['x','y']], labels=[[1,1,0,],[1,0,1,]])
df = pd.DataFrame(np.random.randn(2,3), columns=midx)

In [2]: df
Out[2]:
        one                zero
          y         x         y
0  0.785806 -0.679039  0.513451
1 -0.337862 -0.350690 -1.423253
```

Si vous souhaitez modifier les colonnes en colonnes standard (pas MultiIndex), renommez simplement les colonnes.

```
df.columns = ['A','B','C']
In [3]: df
Out[3]:
          A         B         C
0  0.785806 -0.679039  0.513451
1 -0.337862 -0.350690 -1.423253
```

Comment changer les colonnes standard en MultiIndex

Commencez avec un DataFrame standard

```
df = pd.DataFrame(np.random.randn(2,3), columns=['a','b','c'])

In [91]: df
Out[91]:
          a         b         c
0 -0.911752 -1.405419 -0.978419
1  0.603888 -1.187064 -0.035883
```

Pour passer à MultiIndex, créez un objet `MultiIndex` et assignez-le à `df.columns` .

```
midx = pd.MultiIndex(levels=[['zero', 'one'], ['x','y']], labels=[[1,1,0,],[1,0,1,]])
df.columns = midx

In [94]: df
Out[94]:
            one                zero
              y         x         y
    0 -0.911752 -1.405419 -0.978419
    1  0.603888 -1.187064 -0.035883
```

Colonnes MultiIndex

MultiIndex peut également être utilisé pour créer des DataFrames avec des colonnes à plusieurs niveaux. Utilisez simplement le mot-clé `columns` dans la commande DataFrame.

```
midx = pd.MultiIndex(levels=[['zero', 'one'], ['x','y']], labels=[[1,1,0,],[1,0,1,]])
df = pd.DataFrame(np.random.randn(6,4), columns=midx)

In [86]: df
Out[86]:
        one                zero
          y         x         y
0  0.625695  2.149377  0.006123
1 -1.392909  0.849853  0.005477
```

Afficher tous les éléments de l'index

Pour afficher tous les éléments de l'index, modifiez les options d'impression qui «sparsifie» l'affichage du MultiIndex.

```
pd.set_option('display.multi_sparse', False)
df.groupby(['A','B']).mean()
# Output:
#         C
# A B
# a 1   107
# a 2   102
# a 3   115
# b 5    92
# b 8    98
# c 2    87
# c 4   104
# c 9   123
```

Lire MultiIndex en ligne: https://riptutorial.com/fr/pandas/topic/3840/multiindex

Chapitre 26: Obtenir des informations sur les DataFrames

Examples

Obtenir des informations DataFrame et l'utilisation de la mémoire

Pour obtenir des informations de base sur un DataFrame, y compris les noms de colonne et les types de données:

```
import pandas as pd

df = pd.DataFrame({'integers': [1, 2, 3],
                   'floats': [1.5, 2.5, 3],
                   'text': ['a', 'b', 'c'],
                   'ints with None': [1, None, 3]})

df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 0 to 2
Data columns (total 4 columns):
floats            3 non-null float64
integers          3 non-null int64
ints with None    2 non-null float64
text              3 non-null object
dtypes: float64(2), int64(1), object(1)
memory usage: 120.0+ bytes
```

Pour obtenir l'utilisation de la mémoire du DataFrame:

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 0 to 2
Data columns (total 4 columns):
floats            3 non-null float64
integers          3 non-null int64
ints with None    2 non-null float64
text              3 non-null object
dtypes: float64(2), int64(1), object(1)
memory usage: 234.0 bytes
```

Liste des noms de colonnes DataFrame

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
```

Pour répertorier les noms de colonne dans un DataFrame:

```
>>> list(df)
['a', 'b', 'c']
```

Cette méthode de compréhension de liste est particulièrement utile lorsque vous utilisez le débogueur:

```
>>> [c for c in df]
['a', 'b', 'c']
```

C'est le long chemin:

```
sampledf.columns.tolist()
```

Vous pouvez également les imprimer en tant qu'index au lieu d'une liste (cela ne sera pas très visible pour les cadres de données comportant de nombreuses colonnes):

```
df.columns
```

Les différentes statistiques du Dataframe.

```
import pandas as pd
df = pd.DataFrame(np.random.randn(5, 5), columns=list('ABCDE'))
```

Générer diverses statistiques récapitulatives. Pour les valeurs numériques, le nombre de valeurs non NA / null (`count`), la moyenne (`mean`), l'écart type `std` et les valeurs connues sous le nom de résumé à cinq chiffres :

- `min` : minimum (plus petite observation)
- `25%` : quartile inférieur ou premier quartile (Q1)
- `50%` : médiane (valeur intermédiaire, Q2)
- `75%` : quartile supérieur ou troisième quartile (Q3)
- `max` : maximum (plus grande observation)

```
>>> df.describe()
              A         B         C         D         E
count  5.000000  5.000000  5.000000  5.000000  5.000000
mean  -0.456917 -0.278666  0.334173  0.863089  0.211153
std    0.925617  1.091155  1.024567  1.238668  1.495219
min   -1.494346 -2.031457 -0.336471 -0.821447 -2.106488
25%   -1.143098 -0.407362 -0.246228 -0.087088 -0.082451
50%   -0.536503 -0.163950 -0.004099  1.509749  0.313918
75%    0.092630  0.381407  0.120137  1.822794  1.060268
max    0.796729  0.828034  2.137527  1.891436  1.870520
```

Lire Obtenir des informations sur les DataFrames en ligne:
https://riptutorial.com/fr/pandas/topic/6697/obtenir-des-informations-sur-les-dataframes

Chapitre 27: Outils de calcul

Examples

Rechercher la corrélation entre les colonnes

Supposons que vous ayez un DataFrame de valeurs numériques, par exemple:

```
df = pd.DataFrame(np.random.randn(1000, 3), columns=['a', 'b', 'c'])
```

alors

```
>>> df.corr()
          a         b         c
a  1.000000  0.018602  0.038098
b  0.018602  1.000000 -0.014245
c  0.038098 -0.014245  1.000000
```

trouvera la corrélation de Pearson entre les colonnes. Notez que la diagonale est 1, car chaque colonne est (évidemment) entièrement corrélée à elle-même.

`pd.DataFrame.correlation` un paramètre de `method` facultatif, spécifiant l'algorithme à utiliser. La valeur par défaut est `pearson` . Pour utiliser la corrélation de Spearman, par exemple, utilisez

```
>>> df.corr(method='spearman')
          a         b         c
a  1.000000  0.007744  0.037209
b  0.007744  1.000000 -0.011823
c  0.037209 -0.011823  1.000000
```

Lire Outils de calcul en ligne: https://riptutorial.com/fr/pandas/topic/5620/outils-de-calcul

Chapitre 28: Outils Pandas IO (lecture et sauvegarde de fichiers)

Remarques

La documentation officielle des pandas comprend une page sur IO Tools avec une liste de fonctions pertinentes pour lire et écrire dans des fichiers, ainsi que des exemples et des paramètres communs.

Examples

Lecture du fichier csv dans DataFrame

Exemple de lecture du fichier `data_file.csv` tel que:

Fichier:

```
index,header1,header2,header3
1,str_data,12,1.4
3,str_data,22,42.33
4,str_data,2,3.44
2,str_data,43,43.34

7, str_data, 25, 23.32
```

Code:

```
pd.read_csv('data_file.csv')
```

Sortie:

```
   index    header1  header2  header3
0      1   str_data       12     1.40
1      3   str_data       22    42.33
2      4   str_data        2     3.44
3      2   str_data       43    43.34
4      7   str_data       25    23.32
```

Quelques arguments utiles:

- **`sep`** Le séparateur de champ par défaut est une virgule `,`. Utilisez cette option si vous avez besoin d'un délimiteur différent, par exemple `pd.read_csv('data_file.csv', sep=';')`

- **`index_col`** Avec `index_col = n` (`n` entier) vous dire pandas géants à utiliser la colonne `n` pour indexer le dataframe. Dans l'exemple ci-dessus:

```
pd.read_csv('data_file.csv',  index_col=0)
```

Sortie:

```
          header1  header2  header3
index
 1       str_data       12     1.40
 3       str_data       22    42.33
 4       str_data        2     3.44
 2       str_data       43    43.34
 7       str_data       25    23.32
```

- **`skip_blank_lines`** Par défaut, les lignes vides sont ignorées. Utilisez `skip_blank_lines=False` pour inclure les lignes vides (elles seront remplies avec les valeurs `NaN`)

```
pd.read_csv('data_file.csv',  index_col=0,skip_blank_lines=False)
```

Sortie:

```
          header1  header2  header3
index
 1       str_data       12     1.40
 3       str_data       22    42.33
 4       str_data        2     3.44
 2       str_data       43    43.34
NaN           NaN      NaN      NaN
 7       str_data       25    23.32
```

- **`parse_dates`** Utilisez cette option pour analyser les données de date.

Fichier:

```
date_begin;date_end;header3;header4;header5
1/1/2017;1/10/2017;str_data;1001;123,45
2/1/2017;2/10/2017;str_data;1001;67,89
3/1/2017;3/10/2017;str_data;1001;0
```

Code pour analyser les colonnes `0` et `1` tant que dates:

```
pd.read_csv('f.csv', sep=';', parse_dates=[0,1])
```

Sortie:

```
  date_begin   date_end   header3  header4 header5
0 2017-01-01 2017-01-10  str_data     1001  123,45
1 2017-02-01 2017-02-10  str_data     1001   67,89
2 2017-03-01 2017-03-10  str_data     1001       0
```

Par défaut, le format de date est déduit. Si vous souhaitez spécifier un format de date que vous pouvez utiliser par exemple

```
dateparse = lambda x: pd.datetime.strptime(x, '%d/%m/%Y')
pd.read_csv('f.csv', sep=';',parse_dates=[0,1],date_parser=dateparse)
```

Sortie:

```
  date_begin   date_end   header3  header4 header5
0 2017-01-01 2017-10-01  str_data     1001  123,45
1 2017-01-02 2017-10-02  str_data     1001   67,89
2 2017-01-03 2017-10-03  str_data     1001       0
```

Vous trouverez plus d'informations sur les paramètres de la fonction dans la documentation officielle .

Enregistrement de base dans un fichier csv

```
raw_data = {'first_name': ['John', 'Jane', 'Jim'],
            'last_name': ['Doe', 'Smith', 'Jones'],
            'department': ['Accounting', 'Sales', 'Engineering'],}
df = pd.DataFrame(raw_data,columns=raw_data.keys())
df.to_csv('data_file.csv')
```

Analyse des dates lors de la lecture de csv

Vous pouvez spécifier une colonne contenant des dates afin que les pandas les analysent automatiquement lors de la lecture du fichier csv

```
pandas.read_csv('data_file.csv', parse_dates=['date_column'])
```

Feuille de calcul à dictée de DataFrames

```
with pd.ExcelFile('path_to_file.xls) as xl:
    d = {sheet_name: xl.parse(sheet_name) for sheet_name in xl.sheet_names}
```

Lire une fiche spécifique

```
pd.read_excel('path_to_file.xls', sheetname='Sheet1')
```

Il existe de nombreuses options d'analyse pour `read_excel` *(similaire aux options de* `read_csv` *.*

```
pd.read_excel('path_to_file.xls',
              sheetname='Sheet1', header=[0, 1, 2],
              skiprows=3, index_col=0)  # etc.
```

Test read_csv

```
import pandas as pd
import io

temp=u"""index; header1; header2; header3
1; str_data; 12; 1.4
3; str_data; 22; 42.33
4; str_data; 2; 3.44
2; str_data; 43; 43.34
7; str_data; 25; 23.32"""
#after testing replace io.StringIO(temp) to filename
df = pd.read_csv(io.StringIO(temp),
                 sep = ';',
                 index_col = 0,
                 skip_blank_lines = True)
print (df)
         header1   header2   header3
index
1       str_data        12      1.40
3       str_data        22     42.33
4       str_data         2      3.44
2       str_data        43     43.34
7       str_data        25     23.32
```

Compréhension de la liste

Tous les fichiers sont dans des `files` . Commencez par créer une liste de DataFrames, puis `concat` les:

```
import pandas as pd
import glob

#a.csv
#a,b
#1,2
#5,8

#b.csv
#a,b
#9,6
#6,4

#c.csv
#a,b
#4,3
#7,0

files = glob.glob('files/*.csv')
dfs = [pd.read_csv(fp) for fp in files]
```

```
#duplicated index inherited from each Dataframe
df = pd.concat(dfs)
print (df)
   a  b
0  1  2
1  5  8
0  9  6
1  6  4
0  4  3
```

```
1  7  0
#'reseting' index
df = pd.concat(dfs, ignore_index=True)
print (df)
   a  b
0  1  2
1  5  8
2  9  6
3  6  4
4  4  3
5  7  0
#concat by columns
df1 = pd.concat(dfs, axis=1)
print (df1)
   a  b  a  b  a  b
0  1  2  9  6  4  3
1  5  8  6  4  7  0
#reset column names
df1 = pd.concat(dfs, axis=1, ignore_index=True)
print (df1)
   0  1  2  3  4  5
0  1  2  9  6  4  3
1  5  8  6  4  7  0
```

Lire en morceaux

```
import pandas as pd

chunksize = [n]
for chunk in pd.read_csv(filename, chunksize=chunksize):
    process(chunk)
    delete(chunk)
```

Enregistrer dans un fichier CSV

Enregistrer avec les paramètres par défaut:

```
df.to_csv(file_name)
```

Ecrire des colonnes spécifiques:

```
df.to_csv(file_name, columns =['col'])
```

Le délimiteur Difault est ',' - pour le changer:

```
df.to_csv(file_name,sep="|")
```

Ecrire sans l'en-tête:

```
df.to_csv(file_name, header=False)
```

Écrivez avec un en-tête donné:

```
df.to_csv(file_name, header = ['A','B','C',...]
```

Pour utiliser un encodage spécifique (par exemple "utf-8"), utilisez l'argument de codage:

df.to_csv (nom_fichier, encoding = 'utf-8')

Analyse des colonnes de date avec read_csv

Date ont toujours un format différent, ils peuvent être analysés en utilisant une fonction parse_dates spécifique.

Ce *input.csv* :

```
2016 06 10 20:30:00    foo
2016 07 11 19:45:30    bar
2013 10 12 4:30:00     foo
```

Peut être analysé comme ceci:

```
mydateparser = lambda x: pd.datetime.strptime(x, "%Y %m %d %H:%M:%S")
df = pd.read_csv("file.csv", sep='\t', names=['date_column', 'other_column'],
parse_dates=['date_column'], date_parser=mydateparser)
```

L' argument *parse_dates* est la colonne à analyser
date_parser est la fonction d'analyse

Lire et fusionner plusieurs fichiers CSV (avec la même structure) en un seul fichier DF

```
import os
import glob
import pandas as pd

def get_merged_csv(flist, **kwargs):
    return pd.concat([pd.read_csv(f, **kwargs) for f in flist], ignore_index=True)

path = 'C:/Users/csvfiles'
fmask = os.path.join(path, '*mask*.csv')

df = get_merged_csv(glob.glob(fmask), index_col=None, usecols=['col1', 'col3'])

print(df.head())
```

Si vous souhaitez fusionner les fichiers CSV horizontalement (en ajoutant des colonnes), utilisez `axis=1` lors de l'appel de la fonction `pd.concat()` :

```
def merged_csv_horizontally(flist, **kwargs):
    return pd.concat([pd.read_csv(f, **kwargs) for f in flist], axis=1)
```

Lire le fichier cvs dans un bloc de données pandas lorsqu'il n'y a pas de ligne d'en-tête

Si le fichier ne contient pas de ligne d'en-tête,

Fichier:

```
1;str_data;12;1.4
3;str_data;22;42.33
4;str_data;2;3.44
2;str_data;43;43.34

7; str_data; 25; 23.32
```

vous pouvez utiliser les `names` mots-clés pour fournir des noms de colonnes:

```
df = pandas.read_csv('data_file.csv', sep=';', index_col=0,
                     skip_blank_lines=True, names=['a', 'b', 'c'])

df
Out:
            a   b      c
1   str_data  12   1.40
3   str_data  22  42.33
4   str_data   2   3.44
2   str_data  43  43.34
7   str_data  25  23.32
```

Utiliser HDFStore

```
import string
import numpy as np
import pandas as pd
```

générer un échantillon DF avec différents types de dtypes

```
df = pd.DataFrame({
     'int32':    np.random.randint(0, 10**6, 10),
     'int64':    np.random.randint(10**7, 10**9, 10).astype(np.int64)*10,
     'float':    np.random.rand(10),
     'string':   np.random.choice([c*10 for c in string.ascii_uppercase], 10),
     })

In [71]: df
Out[71]:
      float   int32       int64      string
0  0.649978  848354  5269162190  DDDDDDDDDD
1  0.346963  490266  6897476700  OOOOOOOOOO
2  0.035069  756373  6711566750  ZZZZZZZZZZ
3  0.066692  957474  9085243570  FFFFFFFFFF
```

```
4  0.679182  665894  3750794810  MMMMMMMMMM
5  0.861914  630527  6567684430  TTTTTTTTTT
6  0.697691  825704  8005182860  FFFFFFFFFF
7  0.474501  942131  4099797720  QQQQQQQQQQ
8  0.645817  951055  8065980030  VVVVVVVVVV
9  0.083500  349709  7417288920  EEEEEEEEEE
```

faire un plus grand DF (10 * 100.000 = 1.000.000 lignes)

```
df = pd.concat([df] * 10**5, ignore_index=True)
```

créer (ou ouvrir un fichier HDFStore existant)

```
store = pd.HDFStore('d:/temp/example.h5')
```

enregistrer notre bloc de données dans le fichier h5 (HDFStore), en indexant les colonnes [int32, int64, string]:

```
store.append('store_key', df, data_columns=['int32','int64','string'])
```

afficher les détails du HDFStore

```
In [78]: store.get_storer('store_key').table
Out[78]:
/store_key/table (Table(10,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "int32": Int32Col(shape=(), dflt=0, pos=2),
  "int64": Int64Col(shape=(), dflt=0, pos=3),
  "string": StringCol(itemsize=10, shape=(), dflt=b'', pos=4)}
  byteorder := 'little'
  chunkshape := (1724,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "int32": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "int64": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

afficher les colonnes indexées

```
In [80]: store.get_storer('store_key').table.colindexes
Out[80]:
{
    "int32": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "int64": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

close (flush to disk) notre fichier de magasin

```
store.close()
```

Lire le journal d'accès Nginx (plusieurs guillemets)

Pour plusieurs guillemets, utilisez regex à la place de sep:

```
df = pd.read_csv(log_file,
                 sep=r'\s(?=(?:[^"]*"[^"]*")*[^"]*$)(?![^\[]*\])',
                 engine='python',
                 usecols=[0, 3, 4, 5, 6, 7, 8],
                 names=['ip', 'time', 'request', 'status', 'size', 'referer', 'user_agent'],
                 na_values='-',
                 header=None
                   )
```

Lire Outils Pandas IO (lecture et sauvegarde de fichiers) en ligne:
https://riptutorial.com/fr/pandas/topic/2896/outils-pandas-io--lecture-et-sauvegarde-de-fichiers-

Chapitre 29: Pandas Datareader

Remarques

Le lecteur de données Pandas est un sous-package qui permet de créer un cadre de données à partir de diverses sources de données Internet, notamment:

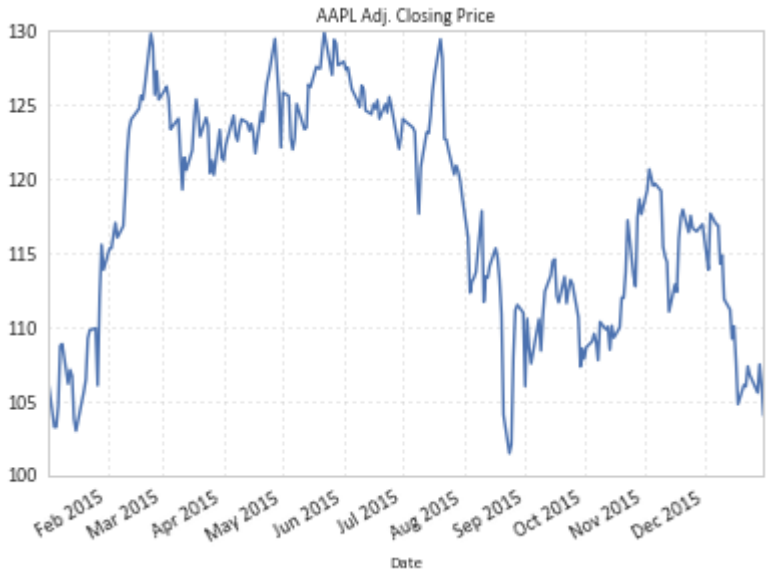
- Yahoo! La finance
- Google Finance
- St.Louis FED (FRED)
- Bibliothèque de Kenneth French
- Banque mondiale
- Google Analytics

Pour plus d'informations, voir ici .

Examples

Exemple de base de Datareader (Yahoo Finance)

```
from pandas_datareader import data

# Only get the adjusted close.
aapl = data.DataReader("AAPL",
                       start='2015-1-1',
                       end='2015-12-31',
                       data_source='yahoo')['Adj Close']

>>> aapl.plot(title='AAPL Adj. Closing Price')
```




```
# Convert the adjusted closing prices to cumulative returns.
returns = aapl.pct_change()
```

```
>>> ((1 + returns).cumprod() - 1).plot(title='AAPL Cumulative Returns')
```

Lecture de données financières (pour plusieurs tickers) dans un panel de pandas - démo

```
from datetime import datetime
import pandas_datareader.data as wb

stocklist = ['AAPL','GOOG','FB','AMZN','COP']

start = datetime(2016,6,8)
end = datetime(2016,6,11)

p = wb.DataReader(stocklist, 'yahoo',start,end)
```

`p` - est un panel de pandas, avec lequel on peut faire des choses drôles:

Voyons ce que nous avons dans notre panel

```
In [388]: p.axes
Out[388]:
[Index(['Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close'], dtype='object'),
 DatetimeIndex(['2016-06-08', '2016-06-09', '2016-06-10'], dtype='datetime64[ns]',
name='Date', freq='D'),
 Index(['AAPL', 'AMZN', 'COP', 'FB', 'GOOG'], dtype='object')]

In [389]: p.keys()
Out[389]: Index(['Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close'], dtype='object')
```

sélection et découpage des données

```
In [390]: p['Adj Close']
Out[390]:
                AAPL        AMZN        COP          FB        GOOG
Date
2016-06-08  98.940002  726.640015  47.490002  118.389999  728.280029
2016-06-09  99.650002  727.650024  46.570000  118.559998  728.580017
```

```
2016-06-10  98.830002  717.909973  44.509998  116.620003  719.409973

In [391]: p['Volume']
Out[391]:
                 AAPL       AMZN        COP          FB       GOOG
Date
2016-06-08  20812700.0  2200100.0  9596700.0  14368700.0  1582100.0
2016-06-09  26419600.0  2163100.0  5389300.0  13823400.0   985900.0
2016-06-10  31462100.0  3409500.0  8941200.0  18412700.0  1206000.0

In [394]: p[:,:,'AAPL']
Out[394]:
                Open       High        Low      Close      Volume  Adj Close
Date
2016-06-08  99.019997  99.559998  98.680000  98.940002  20812700.0  98.940002
2016-06-09  98.500000  99.989998  98.459999  99.650002  26419600.0  99.650002
2016-06-10  98.529999  99.349998  98.480003  98.830002  31462100.0  98.830002

In [395]: p[:,'2016-06-10']
Out[395]:
            Open        High         Low       Close      Volume   Adj Close
AAPL   98.529999   99.349998   98.480003   98.830002  31462100.0   98.830002
AMZN  722.349976  724.979980  714.210022  717.909973   3409500.0  717.909973
COP    45.900002   46.119999   44.259998   44.509998   8941200.0   44.509998
FB    117.540001  118.110001  116.260002  116.620003  18412700.0  116.620003
GOOG  719.469971  725.890015  716.429993  719.409973   1206000.0  719.409973
```

Lire Pandas Datareader en ligne: https://riptutorial.com/fr/pandas/topic/1912/pandas-datareader

Chapitre 30: pd.DataFrame.apply

Examples

pandas.DataFrame.apply Utilisation de base

La méthode pandas.DataFrame.apply () est utilisée pour appliquer une fonction donnée à un `DataFrame` entier --- par exemple, en calculant la racine carrée de chaque entrée d'un `DataFrame` donné ou en sommant sur chaque ligne d'un `DataFrame` pour renvoyer une `Series` .

Ce qui suit est un exemple de base d'utilisation de cette fonction:

```
# create a random DataFrame with 7 rows and 2 columns
df = pd.DataFrame(np.random.randint(0,100,size = (7,2)),
                  columns = ['fst','snd'])

>>> df
   fst  snd
0   40   94
1   58   93
2   95   95
3   88   40
4   25   27
5   62   64
6   18   92

# apply the square root function to each column:
# (this returns a DataFrame where each entry is the sqrt of the entry in df;
# setting axis=0 or axis=1 doesn't make a difference)
>>> df.apply(np.sqrt)
        fst       snd
0  6.324555  9.695360
1  7.615773  9.643651
2  9.746794  9.746794
3  9.380832  6.324555
4  5.000000  5.196152
5  7.874008  8.000000
6  4.242641  9.591663

# sum across the row (axis parameter now makes a difference):
>>> df.apply(np.sum, axis=1)
0    134
1    151
2    190
3    128
4     52
5    126
6    110
dtype: int64

>>> df.apply(np.sum)
fst    386
snd    505
dtype: int64
```

Lire pd.DataFrame.apply en ligne: https://riptutorial.com/fr/pandas/topic/7024/pd-dataframe-apply

Chapitre 31: Rééchantillonnage

Examples

Sous-échantillonnage et suréchantillonnage

```
import pandas as pd
import numpy as np

np.random.seed(0)
rng = pd.date_range('2015-02-24', periods=10, freq='T')
df = pd.DataFrame({'Val' : np.random.randn(len(rng))}, index=rng)
print (df)
                         Val
2015-02-24 00:00:00  1.764052
2015-02-24 00:01:00  0.400157
2015-02-24 00:02:00  0.978738
2015-02-24 00:03:00  2.240893
2015-02-24 00:04:00  1.867558
2015-02-24 00:05:00 -0.977278
2015-02-24 00:06:00  0.950088
2015-02-24 00:07:00 -0.151357
2015-02-24 00:08:00 -0.103219
2015-02-24 00:09:00  0.410599
```

```
#downsampling with aggregating sum
print (df.resample('5Min').sum())
                         Val
2015-02-24 00:00:00  7.251399
2015-02-24 00:05:00  0.128833

#5Min is same as 5T
print (df.resample('5T').sum())
                         Val
2015-02-24 00:00:00  7.251399
2015-02-24 00:05:00  0.128833

#upsampling and fill NaN values method forward filling
print (df.resample('30S').ffill())
                         Val
2015-02-24 00:00:00  1.764052
2015-02-24 00:00:30  1.764052
2015-02-24 00:01:00  0.400157
2015-02-24 00:01:30  0.400157
2015-02-24 00:02:00  0.978738
2015-02-24 00:02:30  0.978738
2015-02-24 00:03:00  2.240893
2015-02-24 00:03:30  2.240893
2015-02-24 00:04:00  1.867558
2015-02-24 00:04:30  1.867558
2015-02-24 00:05:00 -0.977278
2015-02-24 00:05:30 -0.977278
2015-02-24 00:06:00  0.950088
2015-02-24 00:06:30  0.950088
2015-02-24 00:07:00 -0.151357
2015-02-24 00:07:30 -0.151357
```

```
2015-02-24 00:08:00 -0.103219
2015-02-24 00:08:30 -0.103219
2015-02-24 00:09:00  0.410599
```

Lire Rééchantillonnage en ligne: https://riptutorial.com/fr/pandas/topic/2164/reechantillonnage

Chapitre 32: Regroupement des données

Examples

Groupement de base

Grouper par une colonne

En utilisant le DataFrame suivant

```
df = pd.DataFrame({'A': ['a', 'b', 'c', 'a', 'b', 'b'],
                   'B': [2, 8, 1, 4, 3, 8],
                   'C': [102, 98, 107, 104, 115, 87]})

df
# Output:
#    A  B    C
# 0  a  2  102
# 1  b  8   98
# 2  c  1  107
# 3  a  4  104
# 4  b  3  115
# 5  b  8   87
```

Regrouper par colonne A et obtenir la valeur moyenne des autres colonnes:

```
df.groupby('A').mean()
# Output:
#           B    C
# A
# a  3.000000  103
# b  6.333333  100
# c  1.000000  107
```

Grouper par plusieurs colonnes

```
df.groupby(['A','B']).mean()
# Output:
#          C
# A B
# a 2  102.0
#   4  104.0
# b 3  115.0
#   8   92.5
# c 1  107.0
```

Notez qu'après le regroupement de chaque ligne dans le DataFrame résultant est indexé par un tuple ou un MultiIndex (dans ce cas, une paire d'éléments des colonnes A et B).

Pour appliquer plusieurs méthodes d'agrégation à la fois, par exemple pour compter le nombre d'éléments dans chaque groupe et calculer leur moyenne, utilisez la fonction `agg` :

```
df.groupby(['A','B']).agg(['count', 'mean'])
# Output:
#         C
#     count   mean
# A B
# a 2     1  102.0
#   4     1  104.0
# b 3     1  115.0
#   8     2   92.5
# c 1     1  107.0
```

Regroupement des numéros

Pour le DataFrame suivant:

```
import numpy as np
import pandas as pd
np.random.seed(0)
df = pd.DataFrame({'Age': np.random.randint(20, 70, 100),
                   'Sex': np.random.choice(['Male', 'Female'], 100),
                   'number_of_foo': np.random.randint(1, 20, 100)})
df.head()
# Output:

#    Age     Sex  number_of_foo
# 0   64  Female             14
# 1   67  Female             14
# 2   20  Female             12
# 3   23    Male             17
# 4   23  Female             15
```

Group `Age` en trois catégories (ou bacs). Les bacs peuvent être donnés comme

- un entier `n` indiquant le nombre de cases - dans ce cas, les données de la base de données sont divisées en `n` intervalles de taille égale
- une séquence d'entiers indiquant l'extrémité des intervalles ouverts à gauche dans lesquels les données sont divisées - par exemple, des intervalles `bins=[19, 40, 65, np.inf]` crée trois groupes d'âge `(19, 40]` , `(40, 65]` et `(65, np.inf]` .

Pandas attribue automatiquement les versions de chaîne des intervalles comme libellé. Il est également possible de définir des libellés propres en définissant un paramètre `labels` sous la forme d'une liste de chaînes.

```
pd.cut(df['Age'], bins=4)
# this creates four age groups: (19.951, 32.25] < (32.25, 44.5] < (44.5, 56.75] < (56.75, 69]
Name: Age, dtype: category
Categories (4, object): [(19.951, 32.25] < (32.25, 44.5] < (44.5, 56.75] < (56.75, 69]]

pd.cut(df['Age'], bins=[19, 40, 65, np.inf])
# this creates three age groups: (19, 40], (40, 65] and (65, infinity)
Name: Age, dtype: category
Categories (3, object): [(19, 40] < (40, 65] < (65, inf]]
```

Utilisez-le dans `groupby` pour obtenir le nombre moyen de foo:

```
age_groups = pd.cut(df['Age'], bins=[19, 40, 65, np.inf])
df.groupby(age_groups)['number_of_foo'].mean()
# Output:
# Age
# (19, 40]     9.880000
# (40, 65]     9.452381
# (65, inf]    9.250000
# Name: number_of_foo, dtype: float64
```

Tableau croisé des groupes d'âge et du sexe:

```
pd.crosstab(age_groups, df['Sex'])
# Output:
# Sex        Female  Male
# Age
# (19, 40]       22    28
# (40, 65]       18    24
# (65, inf]       3     5
```

Sélection de colonne d'un groupe

Lorsque vous effectuez un groupby, vous pouvez sélectionner une seule colonne ou une liste de colonnes:

```
In [11]: df = pd.DataFrame([[1, 1, 2], [1, 2, 3], [2, 3, 4]], columns=["A", "B", "C"])

In [12]: df
Out[12]:
   A  B  C
0  1  1  2
1  1  2  3
2  2  3  4

In [13]: g = df.groupby("A")

In [14]: g["B"].mean()           # just column B
Out[14]:
A
1    1.5
2    3.0
Name: B, dtype: float64

In [15]: g[["B", "C"]].mean()    # columns B and C
Out[15]:
     B    C
A
1  1.5  2.5
2  3.0  4.0
```

Vous pouvez également utiliser `agg` pour spécifier les colonnes et l'agrégation à effectuer:

```
In [16]: g.agg({'B': 'mean', 'C': 'count'})
Out[16]:
   C    B
A
1  2  1.5
```

```
2  1  3.0
```

Agrégation par taille par rapport au nombre

La différence entre la `size` et le `count` est la suivante:

`size` compte les valeurs `NaN` , le `count` ne le fait pas.

```
df = pd.DataFrame(
        {"Name":["Alice", "Bob", "Mallory", "Mallory", "Bob" , "Mallory"],
         "City":["Seattle", "Seattle", "Portland", "Seattle", "Seattle", "Portland"],
         "Val": [4, 3, 3, np.nan, np.nan, 4]})

df
# Output:
#        City     Name  Val
# 0   Seattle    Alice  4.0
# 1   Seattle      Bob  3.0
# 2  Portland  Mallory  3.0
# 3   Seattle  Mallory  NaN
# 4   Seattle      Bob  NaN
# 5  Portland  Mallory  4.0


df.groupby(["Name", "City"])['Val'].size().reset_index(name='Size')
# Output:
#       Name      City  Size
# 0    Alice   Seattle     1
# 1      Bob   Seattle     2
# 2  Mallory  Portland     2
# 3  Mallory   Seattle     1

df.groupby(["Name", "City"])['Val'].count().reset_index(name='Count')
# Output:
#       Name      City  Count
# 0    Alice   Seattle      1
# 1      Bob   Seattle      1
# 2  Mallory  Portland      2
# 3  Mallory   Seattle      0
```

Groupes d'agrégation

```
In [1]: import numpy as np
In [2]: import pandas as pd

In [3]: df = pd.DataFrame({'A': list('XYZXYZXYZX'), 'B': [1, 2, 1, 3, 1, 2, 3, 3, 1, 2],
                           'C': [12, 14, 11, 12, 13, 14, 16, 12, 10, 19]})

In [4]: df.groupby('A')['B'].agg({'mean': np.mean, 'standard deviation': np.std})
Out[4]:
   standard deviation      mean
A
X            0.957427  2.250000
Y            1.000000  2.000000
Z            0.577350  1.333333
```

Pour plusieurs colonnes:

```
In [5]: df.groupby('A').agg({'B': [np.mean, np.std], 'C': [np.sum, 'count']})
Out[5]:
    C               B
  sum count      mean       std
A
X  59     4  2.250000  0.957427
Y  39     3  2.000000  1.000000
Z  35     3  1.333333  0.577350
```

Exporter des groupes dans des fichiers différents

Vous pouvez effectuer une itération sur l'objet renvoyé par `groupby()` . L'itérateur contient des tuples `(Category, DataFrame)` .

```
# Same example data as in the previous example.
import numpy as np
import pandas as pd
np.random.seed(0)
df = pd.DataFrame({'Age': np.random.randint(20, 70, 100),
                   'Sex': np.random.choice(['Male', factor'Female'], 100),
                   'number_of_foo': np.random.randint(1, 20, 100)})

# Export to Male.csv and Female.csv files.
for sex, data in df.groupby('Sex'):
    data.to_csv("{}.csv".format(sex))
```

utiliser transformation pour obtenir des statistiques au niveau du groupe tout en préservant le cadre de données d'origine

Exemple:

```
df = pd.DataFrame({'group1' :  ['A', 'A', 'A', 'A',
                                'B', 'B', 'B', 'B'],
                   'group2' :  ['C', 'C', 'C', 'D',
                                'E', 'E', 'F', 'F'],
                   'B'      :  ['one', np.NaN, np.NaN, np.NaN,
                                 np.NaN, 'two', np.NaN, np.NaN],
                   'C'      :  [np.NaN, 1, np.NaN, np.NaN,
                               np.NaN, np.NaN, np.NaN, 4]})

 df
Out[34]:
     B    C group1 group2
0  one  NaN      A      C
1  NaN  1.0      A      C
2  NaN  NaN      A      C
3  NaN  NaN      A      D
4  NaN  NaN      B      E
5  two  NaN      B      E
6  NaN  NaN      B      F
7  NaN  4.0      B      F
```

Je veux connaître le nombre d'observations non manquantes de B pour chaque combinaison de

`group1` et de `group2` . `groupby.transform` est une fonction très puissante qui fait exactement cela.

```
df['count_B']=df.groupby(['group1','group2']).B.transform('count')

df
Out[36]:
     B    C group1 group2  count_B
0  one  NaN      A      C        1
1  NaN  1.0      A      C        1
2  NaN  NaN      A      C        1
3  NaN  NaN      A      D        0
4  NaN  NaN      B      E        1
5  two  NaN      B      E        1
6  NaN  NaN      B      F        0
7  NaN  4.0      B      F        0
```

Lire Regroupement des données en ligne:
https://riptutorial.com/fr/pandas/topic/1822/regroupement-des-donnees

Chapitre 33: Regroupement des données de séries chronologiques

Examples

Générer des séries chronologiques de nombres aléatoires puis d'échantillon inférieur

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# I want 7 days of 24 hours with 60 minutes each
periods = 7 * 24 * 60
tidx = pd.date_range('2016-07-01', periods=periods, freq='T')
#                     ^                                   ^
#                     |                                   |
#                 Start Date        Frequency Code for Minute
# This should get me 7 Days worth of minutes in a datetimeindex

# Generate random data with numpy.  We'll seed the random
# number generator so that others can see the same results.
# Otherwise, you don't have to seed it.
np.random.seed([3,1415])

# This will pick a number of normally distributed random numbers
# where the number is specified by periods
data = np.random.randn(periods)

ts = pd.Series(data=data, index=tidx, name='HelloTimeSeries')

ts.describe()

count    10080.000000
mean        -0.008853
std          0.995411
min         -3.936794
25%         -0.683442
50%          0.002640
75%          0.654986
max          3.906053
Name: HelloTimeSeries, dtype: float64
```

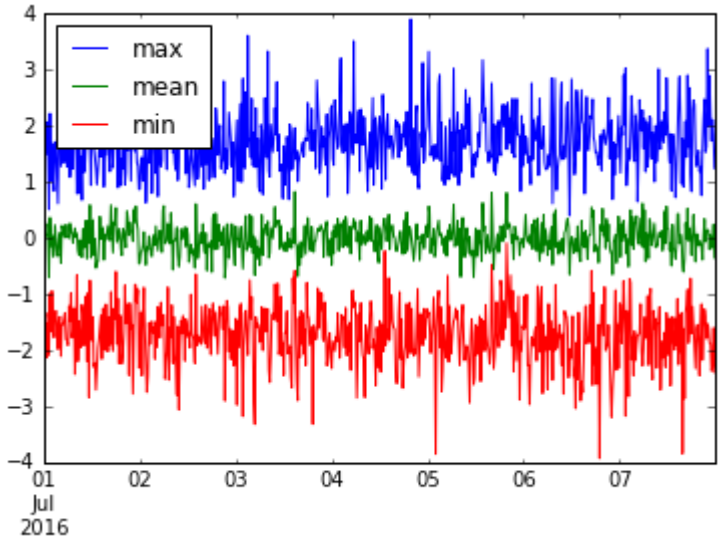
Prenons ces 7 jours de données à la minute et les échantillons à toutes les 15 minutes. Tous les codes de fréquence peuvent être trouvés ici .

```
# resample says to group by every 15 minutes.  But now we need
# to specify what to do within those 15 minute chunks.

# We could take the last value.
ts.resample('15T').last()
```

Ou toute autre chose que nous pouvons faire pour un objet `groupby` , la documentation .

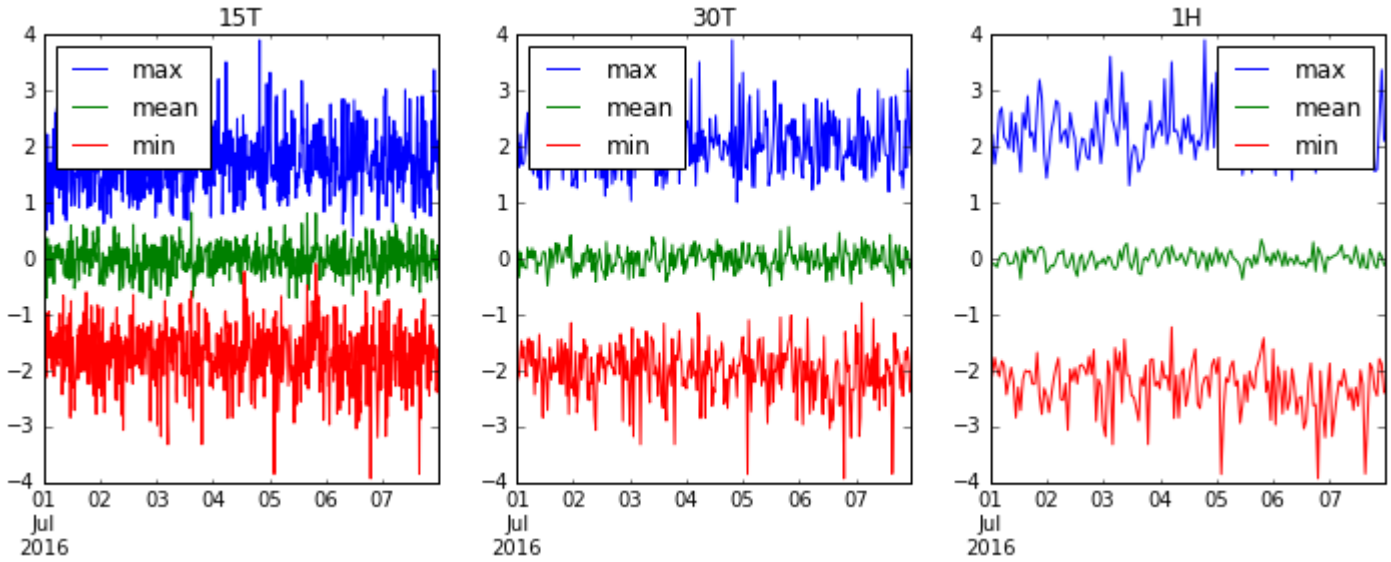
Nous pouvons même agréger plusieurs choses utiles. Tracez les valeurs `min` , `mean` et `max` de cette donnée de `resample('15M')` .

```
ts.resample('15T').agg(['min', 'mean', 'max']).plot()
```



Rééchantillonnons `'15T'` (15 minutes), `'30T'` (demi-heure) et `'1H'` (1 heure) et voyons comment nos données sont plus fluides.

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
for i, freq in enumerate(['15T', '30T', '1H']):
    ts.resample(freq).agg(['max', 'mean', 'min']).plot(ax=axes[i], title=freq)
```



Lire Regroupement des données de séries chronologiques en ligne: https://riptutorial.com/fr/pandas/topic/4747/regroupement-des-donnees-de-series-chronologiques

Chapitre 34: Remodelage et pivotement

Examples

Pivotement simple

Essayez d'abord d'utiliser `pivot` :

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'Name':['Mary', 'Josh','Jon','Lucy', 'Jane', 'Sue'],
                   'Age':[34, 37, 29, 40, 29, 31],
                   'City':['Boston','New York', 'Chicago', 'Los Angeles', 'Chicago',
'Boston'],
                   'Position':['Manager','Programmer','Manager','Manager','Programmer',
'Programmer']},
                    columns=['Name','Position','City','Age'])

print (df)
   Name    Position         City  Age
0  Mary     Manager       Boston   34
1  Josh  Programmer     New York   37
2   Jon     Manager      Chicago   29
3  Lucy     Manager  Los Angeles   40
4  Jane  Programmer      Chicago   29
5   Sue  Programmer       Boston   31

print (df.pivot(index='Position', columns='City', values='Age'))
City        Boston  Chicago  Los Angeles  New York
Position
Manager       34.0     29.0         40.0       NaN
Programmer    31.0     29.0          NaN      37.0
```

Si nécessaire, réinitialisez l'index, supprimez les noms des colonnes et remplissez les valeurs NaN:

```
#pivoting by numbers - column Age
print (df.pivot(index='Position', columns='City', values='Age')
         .reset_index()
         .rename_axis(None, axis=1)
         .fillna(0))

     Position  Boston  Chicago  Los Angeles  New York
0     Manager    34.0     29.0         40.0       0.0
1  Programmer    31.0     29.0          0.0      37.0


#pivoting by strings - column Name
print (df.pivot(index='Position', columns='City', values='Name'))

City       Boston Chicago Los Angeles New York
Position
Manager      Mary     Jon        Lucy     None
Programmer    Sue    Jane        None     Josh
```

Pivoter avec agréger

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'Name':['Mary', 'Jon','Lucy', 'Jane', 'Sue', 'Mary', 'Lucy'],
                   'Age':[35, 37, 40, 29, 31, 26, 28],
                   'City':['Boston', 'Chicago', 'Los Angeles', 'Chicago', 'Boston', 'Boston',
'Chicago'],
                   'Position':['Manager','Manager','Manager','Programmer',
'Programmer','Manager','Manager'],
                    'Sex':['Female','Male','Female','Female', 'Female','Female','Female']},
                    columns=['Name','Position','City','Age','Sex'])

print (df)
   Name    Position         City  Age  Sex
0  Mary     Manager       Boston   35  Female
1   Jon     Manager      Chicago   37  Male
2  Lucy     Manager  Los Angeles   40  Female
3  Jane  Programmer      Chicago   29  Female
4   Sue  Programmer       Boston   31  Female
5  Mary     Manager       Boston   26  Female
6  Lucy     Manager      Chicago   28  Female
```

Si utiliser `pivot` , obtenir une erreur:

```
print (df.pivot(index='Position', columns='City', values='Age'))
```

ValueError: l'index contient des entrées en double, ne peut pas être remodelé

Utilisez `pivot_table` avec la fonction d'agrégation:

```
#default aggfunc is np.mean
print (df.pivot_table(index='Position', columns='City', values='Age'))
City        Boston  Chicago  Los Angeles
Position
Manager       30.5     32.5         40.0
Programmer    31.0     29.0          NaN

print (df.pivot_table(index='Position', columns='City', values='Age', aggfunc=np.mean))
City        Boston  Chicago  Los Angeles
Position
Manager       30.5     32.5         40.0
Programmer    31.0     29.0          NaN
```

Une autre fonction ag

```
print (df.pivot_table(index='Position', columns='City', values='Age', aggfunc=sum))
City        Boston  Chicago  Los Angeles
Position
Manager       61.0     65.0         40.0
Programmer    31.0     29.0          NaN

#lost data !!!
print (df.pivot_table(index='Position', columns='City', values='Age', aggfunc='first'))
City        Boston  Chicago  Los Angeles
```

```
Position
Manager       35.0     37.0         40.0
Programmer    31.0     29.0          NaN
```

Si nécessaire, agréger par colonnes avec `string` valeurs de `string` :

```
print (df.pivot_table(index='Position', columns='City', values='Name'))
```

DataError: aucun type numérique à agréger

Vous pouvez utiliser ces fonctions agrandissantes:

```
print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc='first'))
City       Boston Chicago Los Angeles
Position
Manager      Mary     Jon        Lucy
Programmer    Sue    Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc='last'))
City       Boston Chicago Los Angeles
Position
Manager      Mary    Lucy        Lucy
Programmer    Sue    Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc='sum'))
City          Boston  Chicago Los Angeles
Position
Manager     MaryMary  JonLucy        Lucy
Programmer       Sue     Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc=', '.join))
City            Boston    Chicago Los Angeles
Position
Manager     Mary, Mary  Jon, Lucy        Lucy
Programmer         Sue       Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc=', '.join,
fill_value='-')
         .reset_index()
         .rename_axis(None, axis=1))
     Position      Boston    Chicago Los Angeles
0     Manager  Mary, Mary  Jon, Lucy        Lucy
1  Programmer         Sue       Jane           -
```

Les informations concernant le *sexe* n'ont pas encore été utilisées. Il pourrait être commuté par l'une des colonnes ou être ajouté à un autre niveau:

```
print (df.pivot_table(index='Position', columns=['City','Sex'], values='Age',
aggfunc='first'))

City       Boston Chicago       Los Angeles
Sex        Female  Female  Male      Female
Position
Manager      35.0    28.0  37.0        40.0
Programmer   31.0    29.0   NaN         NaN
```

Plusieurs colonnes peuvent être spécifiées dans l'un des index, colonnes et valeurs des attributs.

```
print (df.pivot_table(index=['Position','Sex'], columns='City', values='Age',
aggfunc='first'))

City               Boston  Chicago  Los Angeles
Position   Sex
Manager    Female    35.0     28.0         40.0
           Male       NaN     37.0          NaN
Programmer Female    31.0     29.0          NaN
```

Application de plusieurs fonctions d'agrégation

Vous pouvez facilement appliquer plusieurs fonctions pendant un seul pivot:

```
In [23]: import numpy as np

In [24]: df.pivot_table(index='Position', values='Age', aggfunc=[np.mean, np.std])
Out[24]:
                 mean       std
Position
Manager     34.333333  5.507571
Programmer  32.333333  4.163332
```

Parfois, vous souhaiterez peut-être appliquer des fonctions spécifiques à des colonnes spécifiques:

```
In [35]: df['Random'] = np.random.random(6)
In [36]: df
Out[36]:
   Name    Position         City  Age    Random
0  Mary     Manager       Boston   34  0.678577
1  Josh  Programmer     New York   37  0.973168
2   Jon     Manager      Chicago   29  0.146668
3  Lucy     Manager  Los Angeles   40  0.150120
4  Jane  Programmer      Chicago   29  0.112769
5   Sue  Programmer       Boston   31  0.185198

For example, find the mean age, and standard deviation of random by Position:

In [37]: df.pivot_table(index='Position', aggfunc={'Age': np.mean, 'Random': np.std})
Out[37]:
                  Age    Random
Position
Manager     34.333333  0.306106
Programmer  32.333333  0.477219
```

On peut aussi passer une liste de fonctions à appliquer aux colonnes individuelles:

```
In [38]: df.pivot_table(index='Position', aggfunc={'Age': np.mean, 'Random': [np.mean,
np.std]})]
Out[38]:
                  Age    Random
                 mean      mean       std
Position
Manager     34.333333  0.325122  0.306106
```

```
Programmer  32.333333  0.423712  0.477219
```

Empilage et dépilage

```
import pandas as pd
import numpy as np

np.random.seed(0)
tuples = list(zip(*[['bar', 'bar', 'foo', 'foo', 'qux', 'qux'],
                    ['one', 'two', 'one', 'two','one', 'two']]))

idx = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
df = pd.DataFrame(np.random.randn(6, 2), index=idx, columns=['A', 'B'])
print (df)
                     A         B
first second
bar   one     1.764052  0.400157
      two     0.978738  2.240893
foo   one     1.867558 -0.977278
      two     0.950088 -0.151357
qux   one    -0.103219  0.410599
      two     0.144044  1.454274
```

```
print (df.stack())
first  second
bar    one     A    1.764052
               B    0.400157
       two     A    0.978738
               B    2.240893
foo    one     A    1.867558
               B   -0.977278
       two     A    0.950088
               B   -0.151357
qux    one     A   -0.103219
               B    0.410599
       two     A    0.144044
               B    1.454274
dtype: float64

#reset index, rename column name
print (df.stack().reset_index(name='val2').rename(columns={'level_2': 'val1'}))
   first second val1      val2
0    bar    one    A  1.764052
1    bar    one    B  0.400157
2    bar    two    A  0.978738
3    bar    two    B  2.240893
4    foo    one    A  1.867558
5    foo    one    B -0.977278
6    foo    two    A  0.950088
7    foo    two    B -0.151357
8    qux    one    A -0.103219
9    qux    one    B  0.410599
10   qux    two    A  0.144044
11   qux    two    B  1.454274
```

```
print (df.unstack())
              A                   B
```

```
second       one       two       one       two
first
bar     1.764052  0.978738  0.400157  2.240893
foo     1.867558  0.950088 -0.977278 -0.151357
qux    -0.103219  0.144044  0.410599  1.454274
```

`rename_axis` (nouveau dans les `pandas 0.18.0`):

```
#reset index, remove columns names
df1 = df.unstack().reset_index().rename_axis((None,None), axis=1)
#reset MultiIndex in columns with list comprehension
df1.columns = ['_'.join(col).strip('_') for col in df1.columns]
print (df1)
  first     A_one     A_two     B_one     B_two
0   bar  1.764052  0.978738  0.400157  2.240893
1   foo  1.867558  0.950088 -0.977278 -0.151357
2   qux -0.103219  0.144044  0.410599  1.454274
```

Pandas ci-dessous 0.18.0

```
#reset index
df1 = df.unstack().reset_index()
#remove columns names
df1.columns.names = (None, None)
#reset MultiIndex in columns with list comprehension
df1.columns = ['_'.join(col).strip('_') for col in df1.columns]
print (df1)
  first     A_one     A_two     B_one     B_two
0   bar  1.764052  0.978738  0.400157  2.240893
1   foo  1.867558  0.950088 -0.977278 -0.151357
2   qux -0.103219  0.144044  0.410599  1.454274
```

Tabulation croisée

```
import pandas as pd
df = pd.DataFrame({'Sex': ['M', 'M', 'F', 'M', 'F', 'F', 'M', 'M', 'F', 'F'],
                   'Age': [20, 19, 17, 35, 22, 22, 12, 15, 17, 22],
                   'Heart Disease': ['Y', 'N', 'Y', 'N', 'N', 'Y', 'N', 'Y', 'N', 'Y']})

df

  Age Heart Disease Sex
0   20             Y   M
1   19             N   M
2   17             Y   F
3   35             N   M
4   22             N   F
5   22             Y   F
6   12             N   M
7   15             Y   M
8   17             N   F
9   22             Y   F

pd.crosstab(df['Sex'], df['Heart Disease'])

Hearth Disease  N  Y
Sex
```

```
F               2  3
M               3  2
```

En utilisant la notation par points:

```
pd.crosstab(df.Sex, df.Age)

Age  12  15  17  19  20  22  35
Sex
F     0   0   2   0   0   3   0
M     1   1   0   1   1   0   1
```

Obtenir la transposition de DF:

```
pd.crosstab(df.Sex, df.Age).T

Sex  F  M
Age
12   0  1
15   0  1
17   2  0
19   0  1
20   0  1
22   3  0
35   0  1
```

Obtenir des marges ou des cumulatifs:

```
pd.crosstab(df['Sex'], df['Heart Disease'], margins=True)

Heart Disease  N  Y  All
Sex
F              2  3    5
M              3  2    5
All            5  5   10
```

Obtenir la transposition du cumulatif:

```
pd.crosstab(df['Sex'], df['Age'], margins=True).T


Sex  F  M  All
Age
12   0  1    1
15   0  1    1
17   2  0    2
19   0  1    1
20   0  1    1
22   3  0    3
35   0  1    1
All  5  5   10
```

Obtenir des pourcentages:

```
pd.crosstab(df["Sex"],df['Heart Disease']).apply(lambda r: r/len(df), axis=1)
```

```
Heart Disease    N    Y
Sex
F              0.2  0.3
M              0.3  0.2
```

Se cumuler et multiplier par 100:

```
df2 = pd.crosstab(df["Age"],df['Sex'], margins=True ).apply(lambda r: r/len(df)*100, axis=1)

df2

Sex     F     M    All
Age
12    0.0  10.0   10.0
15    0.0  10.0   10.0
17   20.0   0.0   20.0
19    0.0  10.0   10.0
20    0.0  10.0   10.0
22   30.0   0.0   30.0
35    0.0  10.0   10.0
All  50.0  50.0  100.0
```

Suppression d'une colonne de DF (one way):

```
df2[["F","M"]]

Sex     F     M
Age
12    0.0  10.0
15    0.0  10.0
17   20.0   0.0
19    0.0  10.0
20    0.0  10.0
22   30.0   0.0
35    0.0  10.0
All  50.0  50.0
```

Les pandas fondent pour passer du long au long

```
>>> df
   ID  Year  Jan_salary  Feb_salary  Mar_salary
0   1  2016        4500        4200        4700
1   2  2016        3800        3600        4400
2   3  2016        5500        5200        5300

>>> melted_df = pd.melt(df,id_vars=['ID','Year'],
                        value_vars=['Jan_salary','Feb_salary','Mar_salary'],
                        var_name='month',value_name='salary')

>>> melted_df
   ID  Year       month  salary
0   1  2016  Jan_salary    4500
1   2  2016  Jan_salary    3800
2   3  2016  Jan_salary    5500
3   1  2016  Feb_salary    4200
4   2  2016  Feb_salary    3600
```

```
5   3  2016  Feb_salary    5200
6   1  2016  Mar_salary    4700
7   2  2016  Mar_salary    4400
8   3  2016  Mar_salary    5300

>>> melted_['month'] = melted_['month'].str.replace('_salary','')

>>> import calendar
>>> def mapper(month_abbr):
...     # from http://stackoverflow.com/a/3418092/42346
...     d = {v: str(k).zfill(2) for k,v in enumerate(calendar.month_abbr)}
...     return d[month_abbr]

>>> melted_df['month'] = melted_df['month'].apply(mapper)
>>> melted_df
   ID  Year month  salary
0   1  2016    01    4500
1   2  2016    01    3800
2   3  2016    01    5500
3   1  2016    02    4200
4   2  2016    02    3600
5   3  2016    02    5200
6   1  2016    03    4700
7   2  2016    03    4400
8   3  2016    03    5300
```

Fractionner (remodeler) les chaînes CSV dans des colonnes en plusieurs lignes, avec un élément par ligne

```
import pandas as pd

df = pd.DataFrame([{'var1': 'a,b,c', 'var2': 1, 'var3': 'XX'},
                   {'var1': 'd,e,f,x,y', 'var2': 2, 'var3': 'ZZ'}])

print(df)

reshaped = \
(df.set_index(df.columns.drop('var1',1).tolist())
   .var1.str.split(',', expand=True)
   .stack()
   .reset_index()
   .rename(columns={0:'var1'})
   .loc[:, df.columns]
)

print(reshaped)
```

Sortie:

```
        var1  var2 var3
0      a,b,c     1   XX
1  d,e,f,x,y     2   ZZ

  var1  var2 var3
0    a     1   XX
1    b     1   XX
2    c     1   XX
```

```
3    d     2   ZZ
4    e     2   ZZ
5    f     2   ZZ
6    x     2   ZZ
7    y     2   ZZ
```

Lire Remodelage et pivotement en ligne: https://riptutorial.com/fr/pandas/topic/1463/remodelage-et-pivotement

Chapitre 35: Sections transversales de différents axes avec MultiIndex

Examples

Sélection de sections en utilisant .xs

```
In [1]:
import pandas as pd
import numpy as np
arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
          ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
idx_row = pd.MultiIndex.from_arrays(arrays, names=['Row_First', 'Row_Second'])
idx_col = pd.MultiIndex.from_product([['A','B'], ['i', 'ii']],
names=['Col_First','Col_Second'])
df = pd.DataFrame(np.random.randn(8,4), index=idx_row, columns=idx_col)

Out[1]:
Col_First                   A                   B
Col_Second                  i        ii         i        ii
Row_First Row_Second
bar       one        -0.452982 -1.872641  0.248450 -0.319433
          two        -0.460388 -0.136089 -0.408048  0.998774
baz       one         0.358206 -0.319344 -2.052081 -0.424957
          two        -0.823811 -0.302336  1.158968  0.272881
foo       one        -0.098048 -0.799666  0.969043 -0.595635
          two        -0.358485  0.412011 -0.667167  1.010457
qux       one         1.176911  1.578676  0.350719  0.093351
          two         0.241956  1.082138 -0.516898 -0.196605
```

`.xs` accepte un `level` (soit le nom dudit niveau ou un entier), et un `axis` : 0 pour les lignes, 1 pour les colonnes.

`.xs` est disponible pour les deux `pandas.Series` et `pandas.DataFrame` .

Sélection sur les lignes:

```
In [2]: df.xs('two', level='Row_Second', axis=0)
Out[2]:
Col_First          A                   B
Col_Second         i        ii         i        ii
Row_First
bar        -0.460388 -0.136089 -0.408048  0.998774
baz        -0.823811 -0.302336  1.158968  0.272881
foo        -0.358485  0.412011 -0.667167  1.010457
qux         0.241956  1.082138 -0.516898 -0.196605
```

Sélection sur colonnes:

```
In [3]: df.xs('ii', level=1, axis=1)
Out[3]:
```

```
Col_First                   A         B
Row_First Row_Second
bar       one        -1.872641 -0.319433
          two        -0.136089  0.998774
baz       one        -0.319344 -0.424957
          two        -0.302336  0.272881
foo       one        -0.799666 -0.595635
          two         0.412011  1.010457
qux       one         1.578676  0.093351
          two         1.082138 -0.196605
```

`.xs` ne fonctionne que pour la sélection, l'affectation n'est PAS possible (obtenir, pas définir): ¨

```
In [4]: df.xs('ii', level='Col_Second', axis=1) = 0
  File "<ipython-input-10-92e0785187ba>", line 1
    df.xs('ii', level='Col_Second', axis=1) = 0
                                            ^
SyntaxError: can't assign to function call
```

Utilisation de .loc et de trancheurs

Contrairement à la méthode `.xs` , cela vous permet d'attribuer des valeurs. L'indexation à l'aide de segments est disponible depuis la version `0.14.0` .

```
In [1]:
import pandas as pd
import numpy as np
arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
          ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
idx_row = pd.MultiIndex.from_arrays(arrays, names=['Row_First', 'Row_Second'])
idx_col = pd.MultiIndex.from_product([['A','B'], ['i', 'ii']],
names=['Col_First','Col_Second'])
df = pd.DataFrame(np.random.randn(8,4), index=idx_row, columns=idx_col)

Out[1]:
Col_First                   A                   B
Col_Second                  i        ii         i        ii
Row_First Row_Second
bar       one        -0.452982 -1.872641  0.248450 -0.319433
          two        -0.460388 -0.136089 -0.408048  0.998774
baz       one         0.358206 -0.319344 -2.052081 -0.424957
          two        -0.823811 -0.302336  1.158968  0.272881
foo       one        -0.098048 -0.799666  0.969043 -0.595635
          two        -0.358485  0.412011 -0.667167  1.010457
qux       one         1.176911  1.578676  0.350719  0.093351
          two         0.241956  1.082138 -0.516898 -0.196605
```

Sélection sur les lignes :

```
In [2]: df.loc[(slice(None),'two'),:]
Out[2]:
Col_First                   A                   B
Col_Second                  i        ii         i        ii
Row_First Row_Second
bar       two        -0.460388 -0.136089 -0.408048  0.998774
```

```
baz       two        -0.823811 -0.302336  1.158968  0.272881
foo       two        -0.358485  0.412011 -0.667167  1.010457
qux       two         0.241956  1.082138 -0.516898 -0.196605
```

Sélection sur colonnes:

```
In [3]: df.loc[:,(slice(None),'ii')]
Out[3]:
Col_First                   A         B
Col_Second                 ii        ii
Row_First Row_Second
bar       one        -1.872641 -0.319433
          two        -0.136089  0.998774
baz       one        -0.319344 -0.424957
          two        -0.302336  0.272881
foo       one        -0.799666 -0.595635
          two         0.412011  1.010457
qux       one         1.578676  0.093351
          two         1.082138 -0.196605
```

Sélection sur les deux axes ::

```
In [4]: df.loc[(slice(None),'two'),(slice(None),'ii')]
Out[4]:
Col_First                   A         B
Col_Second                 ii        ii
Row_First Row_Second
bar       two        -0.136089  0.998774
baz       two        -0.302336  0.272881
foo       two         0.412011  1.010457
qux       two         1.082138 -0.196605
```

L'affectation fonctionne (contrairement à `.xs`):

```
In [5]: df.loc[(slice(None),'two'),(slice(None),'ii')]=0
        df
Out[5]:
Col_First                   A                   B
Col_Second                  i        ii         i        ii
Row_First Row_Second
bar       one        -0.452982 -1.872641  0.248450 -0.319433
          two        -0.460388  0.000000 -0.408048  0.000000
baz       one         0.358206 -0.319344 -2.052081 -0.424957
          two        -0.823811  0.000000  1.158968  0.000000
foo       one        -0.098048 -0.799666  0.969043 -0.595635
          two        -0.358485  0.000000 -0.667167  0.000000
qux       one         1.176911  1.578676  0.350719  0.093351
          two         0.241956  0.000000 -0.516898  0.000000
```

Lire Sections transversales de différents axes avec MultiIndex en ligne: https://riptutorial.com/fr/pandas/topic/8099/sections-transversales-de-differents-axes-avec-multiindex

Chapitre 36: Séries

Examples

Exemples de création de séries simples

Une série est une structure de données à une dimension. C'est un peu comme un tableau suralimenté ou un dictionnaire.

```
import pandas as pd

s = pd.Series([10, 20, 30])

>>> s
0    10
1    20
2    30
dtype: int64
```

Chaque valeur d'une série a un index. Par défaut, les indices sont des nombres entiers allant de 0 à la longueur de la série moins 1. Dans l'exemple ci-dessus, vous pouvez voir les indices imprimés à gauche des valeurs.

Vous pouvez spécifier vos propres indices:

```
s2 = pd.Series([1.5, 2.5, 3.5], index=['a', 'b', 'c'], name='my_series')

>>> s2
a    1.5
b    2.5
c    3.5
Name: my_series, dtype: float64

s3 = pd.Series(['a', 'b', 'c'], index=list('ABC'))

>>> s3
A    a
B    b
C    c
dtype: object
```

Série avec datetime

```
import pandas as pd
import numpy as np

np.random.seed(0)
rng = pd.date_range('2015-02-24', periods=5, freq='T')
s = pd.Series(np.random.randn(len(rng)), index=rng)
print (s)

2015-02-24 00:00:00    1.764052
```

```
2015-02-24 00:01:00    0.400157
2015-02-24 00:02:00    0.978738
2015-02-24 00:03:00    2.240893
2015-02-24 00:04:00    1.867558
Freq: T, dtype: float64

rng = pd.date_range('2015-02-24', periods=5, freq='T')
s1 = pd.Series(rng)
print (s1)

0   2015-02-24 00:00:00
1   2015-02-24 00:01:00
2   2015-02-24 00:02:00
3   2015-02-24 00:03:00
4   2015-02-24 00:04:00
dtype: datetime64[ns]
```

Quelques astuces sur Series in Pandas

Supposons que nous ayons la série suivante:

```
>>> import pandas as pd
>>> s = pd.Series([1, 4, 6, 3, 8, 7, 4, 5])
>>> s
0    1
1    4
2    6
3    3
4    8
5    7
6    4
7    5
dtype: int64
```

Les suivis sont quelques choses simples qui sont utiles lorsque vous travaillez avec Series:

Pour obtenir la longueur de s:

```
>>> len(s)
8
```

Pour accéder à un élément dans s:

```
>>> s[4]
8
```

Pour accéder à un élément dans s en utilisant l'index:

```
>>> s.loc[2]
6
```

Pour accéder à une sous-série à l'intérieur de s:

```
>>> s[1:3]
1    4
2    6
dtype: int64
```

Pour obtenir une sous-série de s avec des valeurs supérieures à 5:

```
>>> s[s > 5]
2    6
4    8
5    7
dtype: int64
```

Pour obtenir le minimum, le maximum, la moyenne et l'écart type:

```
>>> s.min()
1
>>> s.max()
8
>>> s.mean()
4.75
>>> s.std()
2.2519832529192065
```

Pour convertir le type série en valeur flottante:

```
>>> s.astype(float)
0    1.0
1    4.0
2    6.0
3    3.0
4    8.0
5    7.0
6    4.0
7    5.0
dtype: float64
```

Pour obtenir les valeurs de s comme un tableau numpy:

```
>>> s.values
array([1, 4, 6, 3, 8, 7, 4, 5])
```

Pour faire une copie de s:

```
>>> d = s.copy()
>>> d
0    1
1    4
2    6
3    3
4    8
5    7
6    4
7    5
dtype: int64
```

Application d'une fonction à une série

Pandas fournit un moyen efficace d'appliquer une fonction à chaque élément d'une série et d'obtenir une nouvelle série. Supposons que nous ayons la série suivante:

```
>>> import pandas as pd
>>> s = pd.Series([3, 7, 5, 8, 9, 1, 0, 4])
>>> s
0    3
1    7
2    5
3    8
4    9
5    1
6    0
7    4
dtype: int64
```

et une fonction carrée:

```
>>> def square(x):
...     return x*x
```

Nous pouvons simplement appliquer un carré à chaque élément de s et obtenir une nouvelle série:

```
>>> t = s.apply(square)
>>> t
0     9
1    49
2    25
3    64
4    81
5     1
6     0
7    16
dtype: int64
```

Dans certains cas, il est plus facile d'utiliser une expression lambda:

```
>>> s.apply(lambda x: x ** 2)
0     9
1    49
2    25
3    64
4    81
5     1
6     0
7    16
dtype: int64
```

ou nous pouvons utiliser n'importe quelle fonction intégrée:

```
>>> q = pd.Series(['Bob', 'Jack', 'Rose'])
```

```
>>> q.apply(str.lower)
0     bob
1    jack
2    rose
dtype: object
```

Si tous les éléments de la série sont des chaînes, il existe un moyen plus simple d'appliquer des méthodes de chaîne:

```
>>> q.str.lower()
0     bob
1    jack
2    rose
dtype: object
>>> q.str.len()
0    3
1    4
2    4
```

Lire Séries en ligne: https://riptutorial.com/fr/pandas/topic/1898/series

Chapitre 37: Traiter les variables catégorielles

Examples

Codage à chaud avec `get_dummies ()`

```
>>> df = pd.DataFrame({'Name':['John Smith', 'Mary Brown'],
                       'Gender':['M', 'F'], 'Smoker':['Y', 'N']})
>>> print(df)
```

```
  Gender         Name Smoker
0      M   John Smith      Y
1      F   Mary Brown      N
```

```
>>> df_with_dummies = pd.get_dummies(df, columns=['Gender', 'Smoker'])
>>> print(df_with_dummies)
```

```
          Name  Gender_F  Gender_M  Smoker_N  Smoker_Y
0   John Smith       0.0       1.0       0.0       1.0
1   Mary Brown       1.0       0.0       1.0       0.0
```

Lire Traiter les variables catégorielles en ligne: https://riptutorial.com/fr/pandas/topic/5999/traiter-les-variables-categorielles

Chapitre 38: Travailler avec des séries chronologiques

Examples

Création de séries chronologiques

Voici comment créer une série chronologique simple.

```
import pandas as pd
import numpy as np

# The number of sample to generate
nb_sample = 100

# Seeding to obtain a reproductible dataset
np.random.seed(0)

se = pd.Series(np.random.randint(0, 100, nb_sample),
                  index = pd.date_range(start = pd.to_datetime('2016-09-24'),
                                        periods = nb_sample, freq='D'))
se.head(2)

# 2016-09-24    44
# 2016-09-25    47

se.tail(2)

# 2016-12-31    85
# 2017-01-01    48
```

Indexation partielle des chaînes

Une manière très pratique de sous-définir les séries temporelles consiste à utiliser **l'indexation partielle des chaînes** . Il permet de sélectionner une plage de dates avec une syntaxe claire.

Obtenir des données

Nous utilisons le jeu de données dans l'exemple Création de séries chronologiques

Afficher la tête et la queue pour voir les limites

```
se.head(2).append(se.tail(2))

# 2016-09-24    44
# 2016-09-25    47
# 2016-12-31    85
# 2017-01-01    48
```

Sous-location

Maintenant, nous pouvons sous-estimer par année, par mois ou par jour de manière très intuitive.

Par année

```
se['2017']

# 2017-01-01    48
```

Par mois

```
se['2017-01']

# 2017-01-01    48
```

De jour

```
se['2017-01-01']

# 48
```

Avec une gamme d'année, de mois, de jour selon vos besoins.

```
se['2016-12-31':'2017-01-01']

# 2016-12-31    85
# 2017-01-01    48
```

Les pandas fournissent également une fonction de `truncate` dédiée pour cet usage via les paramètres `after` and `before` - mais je pense que c'est moins clair.

```
se.truncate(before='2017')

# 2017-01-01    48

se.truncate(before='2016-12-30', after='2016-12-31')

# 2016-12-30    13
# 2016-12-31    85
```

Lire Travailler avec des séries chronologiques en ligne:
https://riptutorial.com/fr/pandas/topic/7029/travailler-avec-des-series-chronologiques

Chapitre 39: Types de données

Remarques

Les types ne sont pas natifs des pandas. Ils sont le résultat d'un couplage architectural étroit entre pandas et numpy.

le type d'une colonne ne doit en aucun cas être lié au type python de l'objet contenu dans la colonne.

Nous avons ici un `pd.Series` avec des flottants. Le type sera `float` .

Ensuite, nous utilisons `astype` pour le "lancer" pour objecter.

```
pd.Series([1.,2.,3.,4.,5.]).astype(object)
0    1
1    2
2    3
3    4
4    5
dtype: object
```

Le type dt est maintenant objet, mais les objets de la liste sont toujours flottants. Logique si vous savez que dans Python, tout est un objet, et peut être mis à jour pour objecter.

```
type(pd.Series([1.,2.,3.,4.,5.]).astype(object)[0])
float
```

Ici, nous essayons de "lancer" les flotteurs en chaînes.

```
pd.Series([1.,2.,3.,4.,5.]).astype(str)
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
dtype: object
```

Le type dt est maintenant objet, mais le type des entrées dans la liste est chaîne. C'est parce que `numpy` ne traite pas des chaînes, et agit donc comme si elles n'étaient que des objets et sans aucun souci.

```
type(pd.Series([1.,2.,3.,4.,5.]).astype(str)[0])
str
```

Ne faites pas confiance aux types, ils sont un artefact d'un défaut architectural des pandas. Spécifiez-les comme vous devez, mais ne vous fiez pas à quel type est défini sur une colonne.

Examples

Vérification des types de colonnes

Les types de colonnes peuvent être vérifiés par `.dtypes` atrribute of DataFrames.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': [True, False, True]})

In [2]: df
Out[2]:
   A    B      C
0  1  1.0   True
1  2  2.0  False
2  3  3.0   True

In [3]: df.dtypes
Out[3]:
A      int64
B    float64
C       bool
dtype: object
```

Pour une seule série, vous pouvez utiliser l'attribut `.dtype` .

```
In [4]: df['A'].dtype
Out[4]: dtype('int64')
```

Changer de type

`astype()` méthode `astype()` modifie le type d'une série et renvoie une nouvelle série.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0],
                           'C': ['1.1.2010', '2.1.2011', '3.1.2011'],
                           'D': ['1 days', '2 days', '3 days'],
                           'E': ['1', '2', '3']})
In [2]: df
Out[2]:
   A    B          C       D  E
0  1  1.0   1.1.2010  1 days  1
1  2  2.0   2.1.2011  2 days  2
2  3  3.0   3.1.2011  3 days  3

In [3]: df.dtypes
Out[3]:
A      int64
B    float64
C     object
D     object
E     object
dtype: object
```

Changez le type de la colonne A pour flotter et tapez la colonne B en entier:

```
In [4]: df['A'].astype('float')
```

```
Out[4]:
0    1.0
1    2.0
2    3.0
Name: A, dtype: float64

In [5]: df['B'].astype('int')
Out[5]:
0    1
1    2
2    3
Name: B, dtype: int32
```

`astype()` méthode `astype()` est destinée à la conversion de types spécifiques (c.-à-d. que vous pouvez spécifier `.astype(float64')` , `.astype(float32)` ou `.astype(float16)`). Pour la conversion générale, vous pouvez utiliser `pd.to_numeric` , `pd.to_datetime` et `pd.to_timedelta` .

Changer le type en numérique

`pd.to_numeric` change les valeurs en un type numérique.

```
In [6]: pd.to_numeric(df['E'])
Out[6]:
0    1
1    2
2    3
Name: E, dtype: int64
```

Par défaut, `pd.to_numeric` une erreur si une entrée ne peut pas être convertie en nombre. Vous pouvez modifier ce comportement en utilisant le paramètre `errors` .

```
# Ignore the error, return the original input if it cannot be converted
In [7]: pd.to_numeric(pd.Series(['1', '2', 'a']), errors='ignore')
Out[7]:
0    1
1    2
2    a
dtype: object

# Return NaN when the input cannot be converted to a number
In [8]: pd.to_numeric(pd.Series(['1', '2', 'a']), errors='coerce')
Out[8]:
0    1.0
1    2.0
2    NaN
dtype: float64
```

Si nécessaire, vérifiez que toutes les lignes avec entrée ne peuvent pas être converties en `boolean indexing` numérique avec l' `boolean indexing` avec `isnull` :

```
In [9]: df = pd.DataFrame({'A': [1, 'x', 'z'],
                           'B': [1.0, 2.0, 3.0],
                           'C': [True, False, True]})
```

```
In [10]: pd.to_numeric(df.A, errors='coerce').isnull()
Out[10]:
0    False
1     True
2     True
Name: A, dtype: bool

In [11]: df[pd.to_numeric(df.A, errors='coerce').isnull()]
Out[11]:
   A    B      C
1  x  2.0  False
2  z  3.0   True
```

Changer le type en datetime

```
In [12]: pd.to_datetime(df['C'])
Out[12]:
0   2010-01-01
1   2011-02-01
2   2011-03-01
Name: C, dtype: datetime64[ns]
```

Notez que le 2.1.2011 est converti au 1er février 2011. Si vous souhaitez utiliser le 2 janvier 2011, vous devez utiliser le paramètre `dayfirst` .

```
In [13]: pd.to_datetime('2.1.2011', dayfirst=True)
Out[13]: Timestamp('2011-01-02 00:00:00')
```

Changer le type en timedelta

```
In [14]: pd.to_timedelta(df['D'])
Out[14]:
0   1 days
1   2 days
2   3 days
Name: D, dtype: timedelta64[ns]
```

Sélection de colonnes basées sur dtype

`select_dtypes` méthode `select_dtypes` peut être utilisée pour sélectionner des colonnes basées sur dtype.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': ['a', 'b', 'c'],
                           'D': [True, False, True]})

In [2]: df
Out[2]:
   A    B  C      D
0  1  1.0  a   True
1  2  2.0  b  False
2  3  3.0  c   True
```

Avec `exclude` paramètres `include` et `exclude` , vous pouvez spécifier les types souhaités:

```
# Select numbers
In [3]: df.select_dtypes(include=['number'])  # You need to use a list
Out[3]:
   A    B
0  1  1.0
1  2  2.0
2  3  3.0

# Select numbers and booleans
In [4]: df.select_dtypes(include=['number', 'bool'])
Out[4]:
   A    B      D
0  1  1.0   True
1  2  2.0  False
2  3  3.0   True

# Select numbers and booleans but exclude int64
In [5]: df.select_dtypes(include=['number', 'bool'], exclude=['int64'])
Out[5]:
     B      D
0  1.0   True
1  2.0  False
2  3.0   True
```

Résumé des types

`get_dtype_counts` méthode `get_dtype_counts` peut être utilisée pour voir une décomposition des types de données.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': ['a', 'b', 'c'],
                           'D': [True, False, True]})

In [2]: df.get_dtype_counts()
Out[2]:
bool       1
float64    1
int64      1
object     1
dtype: int64
```

Lire Types de données en ligne: https://riptutorial.com/fr/pandas/topic/2959/types-de-donnees

Chapitre 40: Utiliser .ix, .iloc, .loc, .at et .iat pour accéder à un DataFrame

Examples

Utiliser .iloc

.iloc utilise des entiers pour lire et écrire des données sur un DataFrame.

Tout d'abord, créons un DataFrame:

```
df = pd.DataFrame({'one': [1, 2, 3, 4, 5],
                   'two': [6, 7, 8, 9, 10],
                }, index=['a', 'b', 'c', 'd', 'e'])
```

Ce DataFrame ressemble à:

```
   one  two
a    1    6
b    2    7
c    3    8
d    4    9
e    5   10
```

Maintenant, nous pouvons utiliser .iloc pour lire et écrire des valeurs. Lisons la première ligne, première colonne:

```
print df.iloc[0, 0]
```

Cela va imprimer:

```
1
```

Nous pouvons également définir des valeurs. Permet de définir la deuxième colonne, deuxième ligne à quelque chose de nouveau:

```
df.iloc[1, 1] = '21'
```

Et puis regardez pour voir ce qui s'est passé:

```
print df

   one two
a    1   6
b    2  21
c    3   8
d    4   9
```

```
e    5  10
```

Utiliser .loc

.loc utilise des **étiquettes** pour lire et écrire des données.

Configurons un DataFrame:

```
df = pd.DataFrame({'one': [1, 2, 3, 4, 5],
                   'two': [6, 7, 8, 9, 10],
                  }, index=['a', 'b', 'c', 'd', 'e'])
```

Ensuite, nous pouvons imprimer le DataFrame pour voir la forme:

```
print df
```

Cela va sortir

```
   one  two
a    1    6
b    2    7
c    3    8
d    4    9
e    5   10
```

Nous utilisons les **étiquettes de** colonne et de ligne pour accéder aux données avec .loc. Définissons la ligne 'c', colonne 'two' à la valeur 33:

```
df.loc['c', 'two'] = 33
```

Voici à quoi ressemble maintenant le DataFrame:

```
   one  two
a    1    6
b    2    7
c    3   33
d    4    9
e    5   10
```

À noter que l'utilisation de `df['two'].loc['c'] = 33` peut ne pas signaler un avertissement, et peut même fonctionner, mais l'utilisation de `df.loc['c', 'two']` est garantie. , tandis que le premier n'est pas.

Nous pouvons lire des tranches de données, par exemple

```
print df.loc['a':'c']
```

imprimera les lignes a à c. C'est inclusif.

```
   one  two
a    1    6
b    2    7
c    3    8
```

Et enfin, nous pouvons faire les deux ensemble:

```
print df.loc['b':'d', 'two']
```

Produira les lignes b à c de la colonne "deux". Notez que l'étiquette de colonne n'est pas imprimée.

```
b    7
c    8
d    9
```

Si .loc est fourni avec un argument entier qui n'est pas une étiquette, il retourne à l'indexation entière des axes (le comportement de .iloc). Cela rend possible l'indexation mixte d'étiquettes et d'entiers:

```
df.loc['b', 1]
```

renverra la valeur dans la 2ème colonne (index commençant à 0) dans la ligne 'b':

```
7
```

Lire Utiliser .ix, .iloc, .loc, .at et .iat pour accéder à un DataFrame en ligne: https://riptutorial.com/fr/pandas/topic/7074/utiliser--ix---iloc---loc---at-et--iat-pour-acceder-a-un-dataframe

Chapitre 41: Valeurs de la carte

Remarques

il convient de mentionner que si la valeur de clé n'existe pas alors cela soulèvera `KeyError` , dans ces situations , il peut être préférable d'utiliser la `merge` ou `get` ce qui vous permet de spécifier une valeur par défaut si la clé n'existe pas

Examples

Carte du dictionnaire

A partir d'un dataframe `df` :

```
  U   L
111  en
112  en
112  es
113  es
113  ja
113  zh
114  es
```

Imaginez que vous vouliez ajouter une nouvelle colonne appelée `S` prenant les valeurs du dictionnaire suivant:

```
d = {112: 'en', 113: 'es', 114: 'es', 111: 'en'}
```

Vous pouvez utiliser la `map` pour effectuer une recherche sur les clés en renvoyant les valeurs correspondantes dans une nouvelle colonne:

```
df['S'] = df['U'].map(d)
```

qui retourne:

```
  U   L   S
111  en  en
112  en  en
112  es  en
113  es  es
113  ja  es
113  zh  es
114  es  es
```

Lire Valeurs de la carte en ligne: https://riptutorial.com/fr/pandas/topic/3928/valeurs-de-la-carte

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec les pandas	Alexander, Andy Hayden, ayhan, Bryce Frank, Community, hashcode55, Nikita Pestrov, user2314737
2	Ajout à DataFrame	shahins
3	Analyse: tout rassembler et prendre des décisions	piRSquared
4	Calendriers de vacances	Romain
5	Création de DataFrames	Ahamed Mustafa M, Alexander, ayhan, Ayush Kumar Singh, bernie, Gal Dreiman, Geeklhem, Gorkem Ozkaya, jasimpson, jezrael, JJD, Julien Marrec, MaxU, Merlin, pylang, Romain, SerialDev, user2314737, vaerek, ysearka
6	Données catégoriques	jezrael, Julien Marrec
7	Données décalées et décalées	ASGM
8	Données dupliquées	ayhan, Ayush Kumar Singh, bee-sting, jezrael
9	Données manquantes	Andy Hayden, ayhan, EdChum, jezrael, Zdenek
10	Enregistrer les données pandas dans un fichier csv	amin, bernie, eraoul, Gal Dreiman, maxliving, Musafir Safwan, Nikita Pestrov, Olel Daniel, Stephan
11	Faire jouer les Pandas avec les types de données Python natifs	DataSwede
12	Fusionner, rejoindre et concaténer	ayhan, Josh Garlitos, MaThMaX, MaxU, piRSquared, SerialDev, varunsinghal

13	Gotchas de pandas	vlad.rad
14	Graphes et Visualisations	Ami Tavory, Nikita Pestrov, Scimonster
15	Indexation booléenne des dataframes	firelynx
16	Indexation et sélection de données	amin, Andy Hayden, ayhan, double0darbo, jasimpson, jezrael, Joseph Dasenbrock, MaxU, Merlin, piRSquared, SerialDev, user2314737
17	IO pour Google BigQuery	ayhan, tworec
18	JSON	PinoSan, SerialDev, user2314737
19	Lecture de fichiers dans des pandas DataFrame	Arthur Camara, bee-sting, Corey Petty, Sirajus Salayhin
20	Lire MySQL sur DataFrame	andyabel, rrawat
21	Lire SQL Server vers Dataframe	bernie, SerialDev
22	Manipulation de cordes	ayhan, mnoronha, SerialDev
23	Manipulation simple de DataFrames	Alexander, ayhan, Ayush Kumar Singh, Gal Dreiman, Geeklhem, MaxU, paulo.filip3, R.M., SerialDev, user2314737, ysearka
24	Meta: Guide de documentation	Andy Hayden, ayhan, Stephen Leppik
25	MultiIndex	Andy Hayden, benten, danielhadar, danio, Pedro M Duarte
26	Obtenir des informations sur les DataFrames	Alexander, ayhan, Ayush Kumar Singh, bernie, Romain, ysearka
27	Outils de calcul	Ami Tavory
28	Outils Pandas IO (lecture et sauvegarde de fichiers)	amin, Andy Hayden, bernie, Fabich, Gal Dreiman, jezrael, João Almeida, Julien Spronck, MaxU, Nikita Pestrov, SerialDev, user2314737

29	Pandas Datareader	Alexander, MaxU
30	pd.DataFrame.apply	ptsw, Romain
31	Rééchantillonnage	jezrael
32	Regroupement des données	Andy Hayden, ayhan, danio, Geeklhem, jezrael, ℕ𝕠𝕠B̲I̲F̲, QM.py, Romain, user2314737
33	Regroupement des données de séries chronologiques	ayhan, piRSquared
34	Remodelage et pivotement	Albert Camps, ayhan, bernie, DataSwede, jezrael, MaxU, Merlin
35	Sections transversales de différents axes avec MultiIndex	Julien Marrec
36	Séries	Alexander, daphshez, EdChum, jezrael, shahins
37	Traiter les variables catégorielles	Gorkem Ozkaya
38	Travailler avec des séries chronologiques	Romain
39	Types de données	Andy Hayden, ayhan, firelynx, jezrael
40	Utiliser .ix, .iloc, .loc, .at et .iat pour accéder à un DataFrame	bee-sting, DataSwede, farleytpm
41	Valeurs de la carte	EdChum, Fabio Lamanna