Pandas pour Python : L'aide-mémoire ultime

Pourquoi ce guide ?

Pour tous ceux qui travaillent sur des sujets de data science sous Python, Pandas est incontournable.

Lorsque j'ai commencé à apprendre Pandas, il y a quelques années de cela, je passais mon temps à faire des recherches sur Google pour savoir quel code utiliser. C'était compliqué de mémoriser sa syntaxe.

Je savais que ce que j'avais à faire existait déjà sur le net. Et qu'il ne me restait plus qu'à rechercher l'information.

J'ai fait cela pendant longtemps. Un vrai fainéant !

Et un jour je me suis dit qu'il était dommage de ne pas me construire une liste de commandes que j'utilisais tout le temps et de la rassembler dans un seul document.

Faire cela m'a permis de progresser énormément sur Pandas …

D'abord, j'ai compris que Pandas proposait plusieurs façons pour faire une même chose. Et cela m'a aidé à mieux retenir les commandes que Pandas propose.

Ce document m'a aussi permis de me mettre au défi de mieux retenir le code et la syntaxe. Lorsque j'ai des commandes Pandas à traiter, j'essaie de me souvenir ce que ce document contient. Cet effort de mémoire est très bénéfique dans cet apprentissage.

Et enfin, cela permet de démystifier la prétendue complexité de Pandas. Les commandes indispensables sont contenues dans ce guide. Ni plus, ni moins …

J'espère que ce guide vous sera aussi utile qu'il l'a été pour moi !

Have Fun!

Vincent

Vérifier les versions de Pandas


```
Entrée [ ]: import pandas as pd
```

```
Entrée [ ]: pd.__version__
```

```
Entrée [ ]: pd.show_versions()
```

Créer une série manuellement

Entrée [ ]:
```
import numpy as np
```

Entrée [ ]:
```
s = pd.Series([1, 3, 5, np.nan, 6 ,8])
```

Entrée [ ]:
```
# Créer la série en la nommant
s = pd.Series([1, 3, 5, np.nan, 6 ,8], name = 'serie')
```

Entrée [ ]:
```
# Renommer la série
s.name = 'serie2'
```

Entrée [ ]:
```
# Créer et nommer une série à partir d'une liste, et définir son index
s = pd.Series([1, 3, 5, np.nan, 6 ,8], name='serie',
              index =['zero', 'un', 'deux', 'trois', 'quatre', 'cinq'])
```

Entrée [ ]:
```
s['un']
```

Entrée [ ]:
```
# s.count() > ne compte pas la valeur np.nan
# s.size > toutes les valeurs sont comptées
print(s.count(), ' - ',s.size)
```

Entrée [ ]:
```
# créer manuellement une série de type catégorie, non ordonnée
s = pd.Series (['m', 'l', 'xs', 's', 'xl'], dtype='category')
```

Entrée [ ]:
```
s.cat.ordered # Retourne False
```

Entrée [ ]:
```
# Créer manuellement une série de type catégorie ordonnée
s2 = pd.Series (['m', 'l', 'xs ', 's', 'xl '])
size_type = pd.api.types.CategoricalDtype(categories = ['xs','s','m','l','xl'], ordered=True)
s3 = s2.astype(size_type)
```

Créer un dataframe manuellement

Entrée [ ]:
```
# Manuellement, avec un array numpy et un range de date, en nommant les colonnes
df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list("ABCD"))
# list("ABCD") > ['A', 'B', 'C', 'D']
```

Entrée [ ]:
```
# Manuellement, à partir de dictionnaires - k, v : v est une liste et sera une colonne du dataframe - k sera son nom
df = pd.DataFrame({
    "A": 1.0,
    "B": pd.Timestamp("20130102"),
    "C": pd.Series(1, index = list(range(4)), dtype="float32"),
    "D": np.array([3] * 4, dtype="int32"),
    "E": pd.Categorical(["test","train","test","train"]),
    "G": "foo"})
```

Entrée [ ]:
```
# Manuellement, à partir de dictionnaires - k, v : v est une liste et sera une colonne du dataframe - k sera son nom
import string
import random
df = pd.DataFrame(
    {
        "groupe": ["Groupe {}".format(x + 1) for x in range(5)] * 5,
        "Joueur": random.sample(list(string.ascii_lowercase), 25),
        "Performance": np.random.uniform(0.200, 0.400, 25),
    }
)
```

Entrée [ ]:
```
#Créer un dataframe à partir d'un dictionnaire de dictionnaires,
#dont les clés sont partagées et utilisées pour créer les index du dataframe
df=pd.DataFrame({'Attendance': {0: 60, 1: 100, 2: 80,3: 75, 4: 95},
                 'Name': {0: 'Olivia', 1: 'John', 2: 'Laura',3: 'Ben',4: 'Kevin'},
                 'Obtained Marks': {0: 56, 1: 75, 2: 82, 3: 64, 4: 67}})
```

Créer un dataframe manuellement - suite

Entrée [ ]:
```
# Créer un dataframe à partir de listes de tuples - 1 tuple > 1 ligne
df = pd.DataFrame([
    ("bird", "Falconiformes", 389.0),
    ("bird", "Psittaciformes", 24.0),
    ("mammal", "Carnivora", 80.2),
    ("mammal", "Primates", np.nan),
    ("mammal", "Carnivora", 58)
    ],
    index=["falcon", "parrot", "lion", "monkey", "leopard"],
    columns=("class", "order", "max_speed")
)
```

Entrée [ ]:
```
# Créer un dataframe à partir de listes de listes - 1 liste > 1 ligne
df = pd.DataFrame([
    ["Andre", "directeur", 230],
    ["Barbara", "assistante", 25],
    ["Corinne", "comptable", 35]
    ],
    columns=("nom", "fonction", "km")
)
```

Entrée [ ]:
```
l1 = ["Caroline", "directeur", 230]
l2 = ["Barbara", "assistante", 25]
l3 = ["Karine", "comptable", 35]
df = pd.DataFrame([l1,l2,l3], columns=("nom", "fonction", "kms"))
```

Entrée [ ]:
```
# Créer un dataframe manuellement indexé par des dates
dates = pd.date_range("20190101", periods=6)
df = pd.DataFrame(np.random.randn(6,4), index=dates)
```

Créer un dataframe – à partir d'un fichier CSV

Entrée [ ]:
```
#Créer un dataframe à partir de données dans un fichier CSV
df = pd.read_csv(r"C:\data_folder\data.csv")
#r > raw file, on laisse tel quel les \
```

Entrée [ ]:
```
# Créer un dataframe à partir de données dans un fichier CSV
# Décodage des dates
df = pd.read_csv(r"C:\data_folder\data.csv",parse_dates=True)
```

Entrée [ ]:
```
# Créer un dataframe à partir de données dans un fichier CSV
# Décodage des dates
# Le nom des colonnes est sur la première ligne
df = pd.read_csv(r"C:\data_folder\data.csv",parse_dates=True, index_col=0)
```

Entrée [ ]:
```
# Créer un dataframe à partir de données dans un fichier CSV
# Décodage des dates
# Le nom des colonnes est sur la première ligne
# On ne lt qu'une partie des colonnes
df = pd.read_csv(r"C:\data_folder\data.csv",parse_dates=True, index_col=0, usecols=['City', 'State'])
# ou alors
df = pd.read_csv(r"C:\data_folder\data.csv",parse_dates=True, index_col=0, usecols=[0:4])
```

Entrée [ ]:
```
# On ne lit qu'une partie des lignes
df = pd.read_csv('http://bit.ly/uforeports', nrows=3)
```

Entrée [ ]:
```
# Créer un Dataframe, en spécifiant l'encodage en Latin, en parsant les dates,
# et en pointant que le format des dates commencent par les jours (et non les mois - format américain)
df = pd.read_csv('bikes.csv', sep=';', encoding='latin1', parse_dates=['Date'], dayfirst=True, index_col='Date')
```

Entrée [ ]:
```
# Créer un dataframe en concaténant plusieurs fichiers
from glob import glob
stock_files = sorted(glob('data/stocks*.csv')) #Liste des fichiers avec le même pattern
pd.concat((pd.read_csv(file) for file in stock_files), ignore_index=True)
```

Créer un dataframe – à partir d'un fichier Excel

Entrée [ ]:
```
# Créer un dataframe à partir de données dans un fichier Excel
df = pd.read_excel("path/to/data.csv")
```

Créer un dataframe – à partir de données online

Entrée [ ]:
```
# Créer un dataframe à partir d'un CSV en ligne
import pandas as pd
url = ("https://raw.github.com/pandas-dev/pandas/master/pandas/tests/io/data/csv/tips.csv")
tips = pd.read_csv(url)
```

Entrée [ ]:
```
# Créer un dataframe à partir d'un CSV en ligne
url = 'https://raw.githubusercontent.com/justmarkham/DAT8/master/data/chipotle.tsv'
chipo = pd.read_csv(url, sep = '\t')
```

Entrée [ ]:
```
# Exemples de dataset en ligne
drinks = pd.read_csv('http://bit.ly/drinksbycountry')
movies = pd.read_csv('http://bit.ly/imdbratings')
orders = pd.read_csv('http://bit.ly/chiporders', sep='\t')
stocks = pd.read_csv('http://bit.ly/smallstocks', parse_dates=['Date'])
titanic = pd.read_csv('http://bit.ly/kaggletrain')
ufo = pd.read_csv('http://bit.ly/uforeports', parse_dates=['Time'])
```

Créer un dataframe – à partir du presse papier

Entrée [ ]:
```
# Récupérer facilement des données venant d'Excel ou de Google sheet
df = pd.read_clipboard()
```

Créer un dataframe – récapitulatif

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Créer un dataframe – En copiant un df existant

```
Entrée [ ]: df2 = df.copy()
```

Gérer l'affichage d'un notebook Jupyter

Entrée [ ]:
```
# Afficher le setup du nombre maximal de lignes à afficher
pd.get_option('display.max_rows')
```

Entrée [ ]:
```
# Permettre l'affichage de 40 colonnes du dataframe
pd.options.display.max_columns = 40
pd.set_option('display.max_rows', None) # affichage de toutes les colonnes
pd.reset_option('display.max_rows') # reset du paramètrage
```

Entrée [ ]:
```
# Affichage de plus de caractères dans les colonnes
pd.get_option('display.max_colwidth')
pd.set_option('display.max_colwidth', 1000)
```

Entrée [ ]:
```
# Affichage de la précision des nombres
pd.set_option('display.precision', 2)
```

Entrée [ ]:
```
# Affichage du séparateur de milliers
pd.set_option('display.float_format', '{:,}'.format)
```

Entrée [ ]:
```
# AFfichage de l'ensemble des paramètrages possibles pour l'affichage
import pandas as pd
pd.describe_option()
pd.describe_option('rows')
```

Examiner rapidement un dataframe

Entrée [ ]:
```
# Afficher les 5 premières lignes d'un dataframe
df.head()
# Afficher les 10 premières lignes d'un dataframe
df.head(10)
```

Entrée [ ]:
```
# Afficher les 5 dernières lignes d'un dataframe
df.tail()
# Afficher les 10 dernières lignes d'un dataframe
df.tail(10)
```

Entrée [ ]:
```
# Afficher les index d'un dataframe > index pandas
df.index
# Afficher les colonnes d'un dataframe > Index pandas
df.columns
# Afficher les valeurs d'un dataframe > array numpy
df.values
```

Entrée [ ]:
```
# Afficher le nom des colonnes d'un dataframe en faisant une boucle
for val in df:
   print(val)
```

Entrée [ ]:
```
# Afficher des statistiques des colonnes numériques d'un dataframe
df.describe()
#count, mean, std, min, 25%, 50%, 75%, max
```

Entrée [ ]:
```
# Afficher des informations générales d'un dataframe (type d'index, noms des colonnes, types des colonnes, mémoire utilisée)
df.info()
```

Entrée [ ]:
```
# Afficher le nombre de lignes d'un dataframe
len(df)
```

Entrée [ ]:
```
# Afficher les dimensions d'un dataframe
df.shape
df.shape[0] # nombre de lignes
df.shape[1] # nombre de colonnes
```

Examiner plus précisément un dataframe

Entrée [ ]:
```
# Afficher des statistiques des colonnes numériques d'un dataframe, en changeant les quartiles
df.describe(percentiles=[.01, .3, .99])
#count, mean, std, min, 1%, 30%, 50%, 99%, max
```

Entrée [ ]:
```
# Afficher plus de statistiques des colonnes d'un dataframe, numériques ou non
df.describe(include="all")
#count, unique, top, freq, first, last, mean, std, min, 25%, 50%, 75%, max
```

Entrée [ ]:
```
# Afficher plus de statistiques des colonnes d'un dataframe, en fonction de leur type
df.describe(include=['object'])
```

Entrée [ ]:
```
# Itérer sur l'ensemble des lignes d'un dataframe
for index,row in df.iterrows():
   print(index,row["Col1"])
```

Entrée [ ]:
```
# Afficher les types des colonnes et leur nombre
df.dtypes.value_counts()
df.dtypes.value_counts(normalize=True) #pourcentage
df.get_dtype_counts() # Va être déprécié
```

Entrée [ ]:
```
# Sortir des statistiques sur une série
df.col1.agg(['mean', 'min', 'max'])
```

Entrée [ ]:
```
# Sortir des statistiques sur toutes les séries du dataframe
df.agg(['mean', 'min', 'max'])
```

Entrée [ ]:
```
# Compter le nombre de valeurs d'une série qui satisfont une condition
df.col1.gt(20).count() # combien de valeurs de col1 sont supérieures à 20 ?
df.col1.gt(20).mul(100).mean()  # quel pourcentage de valeurs de col1 sont supérieures à 20 ?
```

Examiner un dataframe en faisant un profiling

Entrée [ ]:
```
import pandas_profiling
pandas_profiling.ProfileReport(df)
```

Ordonner, renommer et supprimer des colonnes

Entrée [ ]:
```python
# Réordonner les colonnes d'un dataframe
new_col_order = ['col3','col1', 'col2']
new_df = df[new_col_order]
```

Entrée [ ]:
```python
# Modifier le nom des colonnes du Dataframe avec un dictionnaire
df.rename( columns = {"col1":"New Col1", "col2":"New Col2" })
# ou alors
df.rename( columns = {"col1":"New Col1", "col2":"New Col2" }, inplace=True) # pour sauvegarder les changements
```

Entrée [ ]:
```python
# Modifier directement le nom des colonnes
df.columns = df.columns.str.replace(' ', '_')
```

Entrée [ ]:
```python
# Modifier directement le nom des colonnes
df.columns = map(str.lower, df.columns)
```

Entrée [ ]:
```python
# Modifier le nom des colonnes à la création du dataframe
df = pd.read_csv("df.csv", header=0, names=new_cols)
```

Entrée [ ]:
```python
# Ajouter un prefixe ou un suffixe aux noms des colonnes
df.add_prefix('X_')
df.add_suffix('_Y')
```

Entrée [ ]:
```python
# Inverser l'ordre des colonnes
df.loc[:, ::-1]
```

Entrée [ ]:
```python
# Supprimer des colonnes d'un dataframe
df.drop(['Col1','Col2'], axis=1, inplace=True) # inplace > sauvegarde des résultats
# ou alors
df.drop(['Col1','Col2'], axis='columns', inplace=True)
```

Entrée [ ]:
```python
# Garder des colonnes que d'un certain type
df.select_dtypes(include=[np.number]).dtypes
```

Modifier l'index des lignes d'un dataframe

Entrée [ ]:
```
# Changer l'index d'une série
data = data.set_index('Col1')
```

Entrée [ ]:
```
# Inverser les lignes d'un dataframe
df.loc[::-1]
```

Entrée [ ]:
```
# Modifier les index Lignes d'un dataframe
df.reindex(index=[])
#Nan si nouveaux index inconnus
```

Entrée [ ]:
```
# Modifier les index Lignes d'un dataframe et mettre 0 si index inconnus
df.reindex(index=[],fill_value=0)
```

Entrée [ ]:
```
# Modifier les index Lignes d'un dataframe et mettre les valeurs du dernier index si index inconnus
df.reindex(index=[],method='ffill')
```

Transformer un dataframe

Entrée [ ]:
```
# Transposer un dataframe
df.T
```

Entrée [ ]:
```
Transformer un dataframe en Array numpy
df.to_numpy()
```

Diminuer la taille d'un dataframe

Entrée [ ]:
```
# Visualiser la taille d'un dataframe
df.info(memory_usage='deep')
```

Entrée [ ]:
```
# Visualiser la taille d'une série du dataframe
df.col1.nbytes                  # taille en mémoire
df.col1.astype('Int16').nbytes  # taille en mémoire aprés transformation
df.memory_usage(deep=True)      # inclue les index et les objets
```

Entrée [ ]:
```
# Modifier certaines Séries chaînes de caractères en Catégories
df['continent'] = df.continent.astype('category')
df.continent.head()           # Visualisation des chaînes de caractères
df.continent.cat.codes.head() # Visualisation des chiffres
df.continent.cat.categories   # Voir le codage des catégories
```

Entrée [ ]:
```
# Modifier certaines Séries chaînes de caractères en Catégories ordonnées
priority_dtype = pd.api.types.CategoricalDtype(categories=['LOW', 'MEDIUM', 'HIGH'], ordered=True)
df['priority'] = df['priority'].astype(priority_dtype)
# ou alors
from pandas.api.types import CategoricalDtype
quality_cat = CategoricalDtype(['good', 'very good', 'excellent'], ordered=True)
df['quality'] = df.quality.astype(quality_cat)
# ou alors
values = pd.Series(sorted(set(city_mpg ))) # Liste unique ordonnée
city_type = pd.CategoricalDtype(categories=values ,ordered=True)
city_mpg.astype(city_type)
```

Entrée [ ]:
```
# Sélection aléatoire d'un échantillon du dataframe
df.sample(n=3, random_state=42)
```

Entrée [ ]:
```
# Séparer le df en 2 groupes aléatoires
df1 = df.sample(frac=0.75, random_state=99)
df2 = df.loc[~df.index.isin(df1.index), :]
```

Changer l’apparence d’un dataframe

Entrée [ ]:
```
format_dict = {'Date':'{:%m/%d/%y}', 'Close':'${:.2f}', 'Volume':'{:,}'}
df.style.format(format_dict)
```

Entrée [ ]:
```
(stocks.style.format(format_dict)
 .hide_index()
 .highlight_min('Close', color='red')
 .highlight_max('Close', color='lightgreen')
)
```

Entrée [ ]:
```
(stocks.style.format(format_dict)
 .hide_index()
 .background_gradient(subset='Volume', cmap='Blues')
)
```

Entrée [ ]:
```
(stocks.style.format(format_dict)
 .hide_index()
 .bar('Volume', color='lightblue', align='zero')
 .set_caption('Stock Prices from October 2016')
)
```

Trier un dataframe

Entrée [ ]:
```
# Trier un dataframe par ses index lignes par ordre croissant
df.sort_index(axis=0, ascending=True)
# ou alors
df.sort_index(axis='index', ascending=True)
```

Entrée [ ]:
```
# Trier un dataframe par ses index lignes par ordre décroissant
df.sort_index(axis=0, ascending=False)
```

Entrée [ ]:
```
# Trier un dataframe par ses colonnes par ordre décroissant (noms des colonnes)
df.sort_index(axis=1, ascending=False)
```

Entrée [ ]:
```
# Trier un dataframe par ses les valeurs d'une de ses colonnes par ordre croissant
df.sort_values(by="C", ascending=True)
```

Entrée [ ]:
```
# Trier un dataframe par les valeurs de plusieurs colonnes
df = df.sort(["col1","col2","col3"],ascending=[1,1,0])
```

Sélectionner les colonnes d'un dataframe par leur nom

Entrée [ ]:
```
# Sélectionner une colonne d'un dataframe par son nom et l'affecter à une série
s = df["Col1"]
# ou alors
s = df.Col1 # il faut que le nom ne comporte pas d'espace
```

Entrée [ ]:
```
# Sélectionner une colonne d'un dataframe par son nom et l'affecter à un Dataframe
s = df[["Col1"]]
```

Entrée [ ]:
```
# Sélectionner plusieurs colonnes d'un dataframe par leurs noms et les affecter à un dataframe
df[["Col1","Col2"]]
```

Entrée [ ]:
```
# Sélectionner plusieurs colonnes d'un dataframe par leurs noms et les affecter à un dataframe
df.loc[:, ["Col1","Col2"]]
```

Entrée [ ]:
```
# Sélectionner plusieurs colonnes d'un dataframe appartenant à un range et les affecter à un dataframe
df.loc[:, "Col1":"Col2"]
```

Entrée [ ]:
```
# Sélectionner les colonnes du dataframe qui sont dans une liste
df.filter(items=['col1', 'col2'])
```

Entrée [ ]:
```
# Sélectionner les colonnes du dataframe dont le nom respecte une regex
df.filter(regex='e$', axis=1)
```

Entrée [ ]:
```
# Sélectionner les colonnes du dataframe dont le nom contient une chaine de caractère
df.filter(like='2018', axis=1)
```

Sélectionner les colonnes d'un dataframe par leur position

Entrée [ ]:
```
# Sélectionner plusieurs colonnes d'un dataframe par leurs positions
df.iloc[:,1:3]
```

Entrée [ ]:
```
# Sélectionner la dernière colonne d'un dataframe
df.iloc[:,-1]
```

Sélectionner les colonnes d'un dataframe par leur type

Entrée [ ]:
```
# Sélectionner les colonnes d'un certain type
df.select_dtypes(include=['int']).head()
df.select_dtypes(include=['number', 'object', 'category', 'datetime']).head()
```

Entrée [ ]:
```
# Sélectionner des colonnes en excluant certains types
df.select_dtypes(exclude='number').head()
```

Sélectionner les lignes d'un dataframe par leur index

Entrée [ ]:
```
# Sélectionner des lignes d'un Dataframe par leur index
df[0:3]
df["20210101":"20210131"]
```

Entrée [ ]:
```
# Sélectionner des lignes d'un Dataframe indexé par une date
df['2019']['close']
df.loc['2016':'2019','close']
```

Sélectionner les lignes d'un dataframe par le nom de l'index

Entrée [ ]:
```
# Sélectionner des lignes d'un Dataframe par le nom de leur index
df.loc["idx1"]
df.loc["idx1","idx2"] ou df.loc[["idx1","idx2"]]
```

Entrée [ ]:
```
# Sélectionner une ligne d'un Dataframe par son nom et l'affecter à un dataframe
df.loc[["idx1"]]
```

Sélectionner les lignes d'un dataframe par la position de l'index

Entrée [ ]:
```
# Sélectionner des lignes d'un Dataframe par leur position
df.iloc[1]
df.iloc[1:3,:]
df.iloc[[1,3,4],:]
```

Sélectionner les données suivant un critère logique

Entrée [ ]:
```
# Filtrage booléen
booleen = df["Col1"]>0
df[booleen]
df[booleen].describe()
```

Entrée [ ]:
```
# Sélectionner les lignes qui vérifient une même condition sur une de leur colonne
df[df["Col1"]>0]
```

Entrée [ ]:
```
# Sélectionner les lignes qui vérifient une même condition sur une de leur colonne
df[df["Col1"].isin("un","deux")]
```

Entrée [ ]:
```
# Sélectionner les lignes qui vérifient une même condition sur une de leur colonne de type caractère
df[df["Col1"].str.startswith('J')]
df[df["Col1"].str.contains('y')]
df[~df["Col1"].str.contains('y')]
```

Entrée [ ]:
```
# Sélectionner les lignes qui vérifient une même condition sur une de leur colonne
df.where[df["Col1"]>0]] # Mêmes dimensions que df, contrairement à df[df["Col1"]>0]
```

Entrée [ ]:
```
# Sélectionner les lignes en écrivant une query
df.query('col1>0 and col2 < 0')
```

Entrée [ ]:
```
# Sélectionner les lignes qui contiennent les 5 valeurs extremes d'une colonne
df.nlargest(5,'Col1')
df.nsmallest(5,'Col1')
# Sélectionner les lignes qui contiennent les 5 valeurs minimales d'une colonne et les 5 valeurs maximales d'une autre colonne
df.nlargest(5,'Col1').nsmallest(5,'Col2')
```

Entrée [ ]:
```
# Sélectionner toutes les lignes d'un dataframe qui font partie de catégories les plus/moins représentées
counts = df.col1.value_counts()
df[df.col1.isin(counts.nlargest(3).index)].head()
```

Entrée [ ]:
```
# Sélectionner les lignes qui vérifient plusieurs mêmes conditions sur leurs colonnes (And)
df[(df["col1"]>0) & (df["col2"] < 0)]
```

Entrée [ ]:
```
# Sélectionner les lignes qui vérifient plusieurs mêmes conditions sur leurs colonnes (Or)
df[(df["col1"]>0) | (df["col2"] < 0)]
```

Entrée [ ]:
```
# Sélectionner les lignes qui vérifient plusieurs mêmes conditions sur leurs colonnes (And)
df[ np.logical_and(df["col1"]>0, df["col2"] < 0] # moins fréquent
```

Ajouter une nouvelle colonne

Entrée [ ]:
```
# Ajouter une nouvelle colonne et l'initialiser à zéro
df.loc[:,"nouvelle"] = 0
# ou alors
df["nouvelle"] = 0
```

Entrée [ ]:
```
# Ajouter une nouvelle colonne et l'initialiser à partir des données d'autres colonnes
df.loc[:,"nouvelle"] = df.loc[:,"col1"]  + df.loc[:,"col2"]
df["nouvelle"] = df["col1"]  + df["col2"]
```

Entrée [ ]:
```
# Ajouter une nouvelle colonne et l'initialiser à partir des données d'autres colonnes
# Calculs à partir des méthodes liées aux séries
df["nouvelle"] = df["col1"].add(df["col2"], fill_value=0)
df["nouvelle"] = df["col1"].add(df["col2"], fill_value=0).div(2) # chaining
```

Entrée [ ]:
```
# Ajouter une nouvelle colonne et l'initialiser à partir des données d'autres colonnes avec la méthode where de numpy
df["nouvelle"] = np.where(df["col1"]<10, "Pas Mal","Bien")
```

Entrée [ ]:
```
# Remplacer les valeurs d'une série qui ne sont pas dans le top5 des valeurs les plus représentées par "Autres"
top5 = col1.value_counts().index[:5]
col1.where(col1.isin(top5), other='Autres')
```

Entrée [ ]:
```
# Et si on veut inverser la logique de la cellule du dessus
top5 = col1.value_counts().index[:5]
col1.where(~col1.isin(top5), other='Autres') #~ inverse la logique
# ou alors
col1.mask(col1.isin(top5), other='Autres')   # Même effet que le ~
```

Entrée [ ]:
```
# Garder les valeurs d'une série qui sont dans le top5 des valeurs les plus représentées,
# mettre "TOP10" pour celles qui sont dans le Top10, et "Autres" sinon
top5 = col1.value_counts().index[:5]
top10 = col1.value_counts().index[:10]
col1.where(col1.isin(top5), other='TOP10').where(col1.isin(top10), other='Autres')
# ou alors
import numpy as np
np.select([col1.isin(top5), col1.isin(top10)],[col1, 'Top10'], 'Other')
```

Modifier des colonnes de type string

Entrée [ ]:
```
# Modifier des chaînes de caractères en utilisant des String Methods
df["col"] = df["col"].str.upper()                                  # tout en majuscule
df["col"] = df["col"].str.replace('[', '').str.replace(']', '') # on supprime les chaines "[]"
df["col"] = df["col"].str.replace('$', '').astype(float)         # on supprime le signe $ et on convertit en nombre
```

Entrée [ ]:
```
# Modifier des Strings en nombres en gérant les erreurs - sur une série
pd.to_numeric(df.col1, errors='coerce')                # Nan si erreur
pd.to_numeric(df.col_three, errors='coerce').fillna(0) # 0 si erreur
```

Entrée [ ]:
```
# Modifier des Strings en nombres en gérant les erreurs - sur le dataframe complet
df = df.apply(pd.to_numeric, errors='coerce').fillna(0)
```

Entrée [ ]:
```
# Modification d'un string dans une Série - Remplacement simple
df.col1.replace(to_replace='old',value='new')
# ou alors
df.col1.replace(to_replace={'old':'new'})
```

Entrée [ ]:
```
# Modification d'un string dans une Série - avec les regex
df.col1.replace(to_replace='z.*',value='zanne',regex=True)
make.replace(r'( Fer)ra(r.*)',value=r'\2-other -\1', regex=True)
```

Entrée [ ]:
```
# Modifier plusieurs colonnes String > Nombre
cols = df.columns                                    # choix de la liste des colonnes
df[cols] = df[cols].apply(pd.to_numeric, errors='coerce')
```

Changer le type d'une colonne

Entrée [ ]:
```
# Transformer une colonne en nombre
df["col"] = df["col"].astype(float)
```

Entrée [ ]:
```
# Modifier Plusieurs colonnes
df = df.astype({'col1':'float', 'col2':'float'})
```

Entrée [ ]:
```
# Modifier les valeurs à la création du dataframe
pd.read_csv('df.csv', dtype={'column_x':float})
```

Entrée [ ]:
```
# Vérifier les capacités des différents types (int64, uint8, float16, float64) - Min et Max possibles
np.iinfo('int64')
# iinfo(min = -9223372036854775808 , max =9223372036854775807 , dtype=int64)
np.iinfo('uint8')
# iinfo(min=0, max =255 , dtype=uint8)
np.finfo('float16')
# finfo(resolution =0.001 , min = -6.55040e+04, max =6.55040e+04, dtype=float16)
np.finfo('float64')
# finfo(resolution =1e-15, min = -1.7976931348623157e+308 ,max =1.7976931348623157 e+308 , dtype=float64)
```

Changer le type d'une colonne en catégorie

Entrée [ ]:
```
# Modifier une colonne du Dataframe contenant un nombre fini de modalités par des nombres
data['sex'].map({'male':0, 'female':1}) # map et dictionnaire
# ou alors
data['sex'].replace( ['male','female'] , [0,1])

data['sex'].astype('category').cat.codes
```

Modifier une valeur du dataframe

Entrée [ ]:
```
# Modifier la valeur en se basant sur les noms des index lignes et colonnes
df.at[dates[0],"A"]=0
```

Entrée [ ]:
```
# Modifier la valeur en se basant sur les positions de la ligne et colonne
df.iat[0,1]=0
df.iat[0,1]=np.nan
```

Entrée [ ]:
```
# Modifier dans le dataframe tout un ensemble de valeurs
df.loc[df.col1=='NOT RATED', 'col1'] = np.nan
```

Ajouter une catégorie pour des valeurs numériques

Entrée [ ]:
```
# bornes définies par l'utilisateur et nommage de chaque groupe
pd.cut(titanic.Age, bins=[0, 18, 25, 99], labels=['child', 'young adult', 'adult'])

pd.cut(df.col1, 10)              # 10 groupes avec bornes réparties équitablement
pd.cut(df.col1, [0,10,20,30,40]) # Avec définition des bornes
pd.qcut(df.col1, 10)             # 10 groupes répartis équitablement en nombre (quartiles)
```

Ajouter un classement

Entrée [ ]:
```
col1.rank()              # rang des valeurs identiques > moyenne de leur rang
col1.rank(method='min')  # les valeurs identiques auront le même rang
col1.rank(method='dense') # pas de trous dans les classements
```

Manipuler un dataframe Time Series

Entrée [ ]:
```
# Convertir la Série en format datetime
df['Time'] = pd.to_datetime(df.Time)
df['Time'] = pd.to_datetime(df.Time, format="%d/%m/%y")
df['Time'] = pd.to_datetime(df.Time, infer_datetime_format=True) # Panda devine le bon format de date à appliquer
```

Entrée [ ]:
```
# A partir d'une données de type date, extraire les informations spécifiques liées aux dates
df.Time.dt.hour
df.Time.dt.weekday_name
df.Time.dayofyear.hour
(day.Time.max() - day.Time.min()).days
```

Entrée [ ]:
```
# Filtrer les données de type date
df[df.time_column > pd.datetime_column(2014, 1, 1)]
```

Compter les valeurs manquantes d'un dataframe

Entrée [ ]:
```
# Visualiser les colonnes d'un dataframe et du nombre de valeurs manquantes
df.info()
```

Entrée [ ]:
```
# identifier les colonnes avec des valeurs manquantes et en faire une liste
cols_miss_val = [col for col in df.columns if df[col].isnull().any()]
```

Entrée [ ]:
```
# Visualiser les valeurs NA du dataframe
pd.isna(df)
```

Entrée [ ]:
```
# Visualiser le nombre de NA du dataframe par colonne
pd.isna(df).sum()
# ou alors
df.isnull().sum()
```

Entrée [ ]:
```
# Visualiser le nombre de valeurs non nulles pour une colonne du dataframe
df["col1"].notnull().sum()
df.notnull().sum()
```

Entrée [ ]:
```
# Visualiser le pourcentage de valeurs non nulles pour les colonnes du dataframe
df.notnull().mean()
```

Entrée [ ]:
```
# Visualiser le nombre de NA du dataframe au total
pd.isna(df).sum().sum()
# ou alors
df.isnull().sum().sum()
```

Entrée [ ]:
```
# Visualiser le nombre de NA du dataframe par type de colonne
df.isnull().dtypes.value_counts()
df.isnull().dtypes.value_counts(normalize=True) #pourcentage
```

Entrée [ ]:
```
# Visualiser le nombre de valeurs non NA du dataframe par colonne
df.count()
```

Entrée [ ]:
```
# Visualiser le pourcentage de NA du dataframe par colonne
pd.isna(df).mean()
```

Entrée [ ]:
```
# Les valeurs manquantes sont exclues par défaut du comptage
df.column_x.value_counts()              # exclusion des valeurs manquantes
df.column_x.value_counts(dropna=False)  # inclusion des valeurs manquantes
```

Entrée [ ]:
```
# Créer le dataframe sans tenir compte du filtre des valeurs manquantes
df = pd.read_csv("df.csv", header=0, names=new_cols, na_filter=False)
```

Supprimer les valeurs manquantes d'un dataframe

Entrée [ ]:
```
# Avant de supprimer, vérifier combien de lignes seront supprimées
df.dropna(how="any").shape # Au moins une des valeurs la ligne est manquante
df.dropna(how="all").shape # Toutes les valeurs de la ligne sont manquantes
```

Entrée [ ]:
```
# Supprimer d'un dataframe toutes les lignes qui comportent une valeur NA
df.dropna(how="any") # au moins une
df.dropna(how="all") # Toutes
```

Entrée [ ]:
```
# Supprimer d'un dataframe toutes les lignes qui comportent une valeur NA sur certaines colonnes
df.dropna(subset=['col1', 'col2'],how="any") # au moins une
df.dropna(subset=['col1', 'col2'],how="all") # toutes
```

Entrée [ ]:
```
# Supprimer d'un dataframe toutes les lignes qui comportent une valeur NA
df.dropna(axis=0)
```

Entrée [ ]:
```
# Supprimer les colonnes qui ont plus de 10% de valeurs manquantes
df.dropna(thresh=len(df)*0.9, axis='columns')
```

Remplacer les valeurs manquantes d'un dataframe

Entrée [ ]:
```
# Remplacer dans un dataframe les valeurs NA par une constante
df.fillna(value=0)
```

Entrée [ ]:
```
# Remplacer dans un dataframe les valeurs NA par une constante
df['col1'].fillna(value='defaut',inplace=True)
```

Entrée [ ]:
```
# Remplacer dans un dataframe les valeurs NA par des constantes différentes suivant les colonnes
df = df.fillna({
 'col1': 'missing',
 'col2': '99.999',
 'col3': '999',
 'col4': 'missing',
 'col5': 'missing',
 'col6': '99'
})
```

Entrée [ ]:
```
# Remplacer dans un dataframe les valeurs NA d'une colonne en se basant sur les valeurs qui précèdent
df["col1"].fillna(inplace=True, method='bfill') # Backward fill
# ou alors
df["col1"].bfill(inplace=True)                  # equivalent
```

Entrée [ ]:
```
# Remplacer dans un dataframe les valeurs NA d'une colonne en se basant sur les valeurs qui suivent
df["col1"].fillna(inplace=True, method='ffill') # forward fill
```

Entrée [ ]:
```
# Remplacer dans un dataframe les valeurs NA d'une colonne en interpolant les valeurs
df["col1"].interpolate() #interpolation
```

Retirer les valeurs extrêmes d'un dataframe

Entrée [ ]:
```
# Enlever dans un dataframe les valeurs extrêmes, à partir des quartiles
# Les remplacer par les bornes du quartile
df.col1.clip(lower=df.col1.quantile(.05), upper=df.col1.quantile(.95))
```

Gérer les doublons dans une série

Entrée [ ]:
```
# Trouver les doublons dans une série
df.col1.duplicated().tail()
```

Entrée [ ]:
```
# Compter le nombre de doublons
df.col1.duplicated().sum()
```

Entrée [ ]:
```
# Afficher tous les doublons pour une série
df.loc[col1.duplicated(keep=False), :]
```

Entrée [ ]:
```
# Afficher tous les doublons pour une série (en ignorant première occurrence)
df.loc[col1.duplicated(keep='first'), :]
```

Entrée [ ]:
```
# Afficher tous les doublons pour une série (en ignorant la dernière occurrence)
df.loc[col1.duplicated(keep='last'), :]
```

Entrée [ ]:
```
# Supprimer les doublons pour une série
df.col1.drop_duplicated(keep='first') # Pour chaque groupe dupliqué, on ne garde que la première occurence trouvée
# ou alors
df.col1.drop_duplicated(keep='last') # On ne garde que la dernière occurence trouvée
# ou alors
df.col1.drop_duplicated(keep='false') # On supprime tous les doublons
```

Gérer les doublons dans un dataframe

Entrée [ ]:
```
# Trouver les doublons dans la df
df.duplicated().tail()
```

Entrée [ ]:
```
# Supprimer les doublons pour un dataframe
df.drop_duplicated(keep='first')
# ou alors
df.drop_duplicated(keep='last')
# ou alors
df.drop_duplicated(keep='false')
```

Faire des calculs statistiques sur des colonnes

Entrée [ ]:
```
# Visualiser la moyenne des colonnes (format numérique) du dataframe
df.mean()
```

Entrée [ ]:
```
# Visualiser la somme des colonnes  (format numérique) du dataframe
df.sum()
```

Entrée [ ]:
```
# Visualiser le nombre de valeurs non nulles par colonne du dataframe
df.count()
```

Entrée [ ]:
```
# Visualiser les statistiques sur une colonne
df["Col1"].mean()
df["Col1"].count()
df["Col1"].sum()
```

Entrée [ ]:
```
# Visualiser les différentes valeurs d'une colonne et leur nombre d'apparition
df["Col1"].value_counts()
df["Col1"].value_counts(1) # sous forme de pourcentage
df["Col1"].value_counts(normalize=True) # sous forme de pourcentage
```

Entrée [ ]:
```
# Visualiser les différentes valeurs d'une colonne
df["Col1"].unique()
```

Entrée [ ]:
```
# Visualiser le nombre des différentes valeurs d'une colonne
df["Col1"].nunique()
```

Entrée [ ]:
```
# Calculer la corrélation entre 2 colonnes du dataframe
df[ ['c1','c2'] ].corr()
```

Entrée [ ]:
```
# Visualiser la somme par ligne de toutes les colonnes numériques du dataframe
df.sum(1)
```

Calculer avec la fonction apply

Entrée [ ]:
```
Note : apply peut ne pas être une solution performante ...
```

Entrée [ ]:
```
# Effectuer une somme cumulative sur toutes les colonnes d'un dataframe
df.apply(np.cumsum)
```

Entrée [ ]:
```
# Trouver pour chaque colonne d'un dataframe, la différence entre la valeur maximale et minimale (par colonne)
df.apply(lambda x: x.max() - x.min())
```

Entrée [ ]:
```
# Trouver pour chaque colonne d'un dataframe, l'index qui contient la valeur maximale
df.apply(lambda x:x.idmax())
```

Entrée [ ]:
```
# Calculer la somme par colonne d'un dataframe
result = df.apply(np.sum, axis=0)
```

Entrée [ ]:
```
# Calculer la somme par ligne d'un dataframe
result = df.apply(np.sum, axis=1)
```

Entrée [ ]:
```
# Mettre dans une nouvelle colonne du dataframe la longueur d'une colonne de type caractère
df["nouvelle"] = df["Col2"].apply(len)
```

Entrée [ ]:
```
# Mettre dans une nouvelle colonne du dataframe une chaine en majuscule issue d'une autre colonne
df["nouvelle"] = df["Col2"].apply(str.upper)
```

Entrée [ ]:
```
# Sortir le premier élément d'une liste (split de caractères)
df.col1.str.split(',').apply(lambda x: x[0])
```

Fusionner deux dataframe - Concaténation

Entrée [ ]:
```
# Fusionner 2 dataframes verticalement, qui ont les mêmes colonnes
df = pd.concat([df1,df2], ignore_index=True)
```

Entrée [ ]:
```
# Fusionner 2 dataframes horizontalement, en faisant la jointure avec les index
df = pd.concat([df1,df2], axis=1)
```

Fusionner deux dataframe - Merge

Entrée [ ]:
```
# Merger 2 dataframes avec une clé de jointure commune entre les 2 df (et de même nom)
pd.merge(df_left, df_right, on="key")
```

Entrée [ ]:
```
# Merger 2 dataframes avec une clé de jointure commune entre les 2 df
pd.merge(df_left, df_right, left_on="key1", right_on="key1", how='outer')
```

Entrée [ ]:
```
# Merger 2 Dataframe, en précisant le type de jointure, la clé de jointure et en renommant les colonnes mergées
pd.merge(btc,eth,on='Date',how='inner',suffixes=('_btc','_eth')) # how = 'inner', 'outer'
```

Regrouper et agréger – Group By

Entrée [ ]:
```
# Faire un group by de somme de toutes les colonnes numériques, suivant les valeurs uniques d'une colonne d'un dataframe
df.groupby("Col1").sum()
```

Entrée [ ]:
```
# Faire un group by de moyenne de toutes les colonnes numériques, suivant les valeurs uniques d'une colonne d'un dataframe
df.groupby("Col1").mean()
```

Entrée [ ]:
```
# Faire un group by de somme  de toutes les colonnes numériques, suivant les valeurs de deux colonnes de dataframe
df.groupby(["Col1","Col2"]).sum()
```

Entrée [ ]:
```
# Faire un group by de compte de toutes les colonnes numériques, suivant les valeurs d'une colonne d'un dataframe
df.groupby("Col1").count()
```

Entrée [ ]:
```
# Spécialiser le comptage sur une colonne (ici Col3)
df.groupby(["Col1","Col2"])["Col3"].sum()
```

Entrée [ ]:
```
# Examiner les groupes issus d'un groupby
obj = df.groupby('col1')
obj.groups
for name,group in obj:
    print(name,'contains',group.shape[0],'rows')
obj.get_group('Tier 1')
```

Entrée [ ]:
```
# Faire plusieurs calculs par regroupement
df.groupby('col1').agg([np.mean,np.median])
```

Entrée [ ]:
```
# Faire des regroupements sur plusieurs colonnes et mettre à plat le tableau (pas de multi-index)
df.groupby(['col1','col2'],as_index=False).agg({'col3':np.mean})
```

Entrée [ ]:
```
# Faire des regroupements sur plusieurs colonnes, différents calculs sur la même quantité et mettre à plat le tableau
df.groupby(['co1','col2'],as_index=False).agg({'col3':[np.mean,np.sum,'count']})
```

Entrée [ ]:
```
# Mettre le nom des colonnes à plat avec un group by
tab.columns = tab.columns.map(' '.join) # Mise à plat des colonnes multi-index
```

Entrée [ ]:
```
# Présenter plus simplement un groupby à plusieurs niveaux
titanic.groupby(['Sex', 'Pclass']).Survived.mean().unstack() # Pclass passe de ligne (index niveau2) à colonne
```

Faire des calculs sur des Time Series

Entrée [ ]:
```
# Faire un group by de données, suivant une période temporelle (jour, mois, année, …)
# Sélection d'un range de date (entre deux années)
df.loc['2016':'2019','close'].resample('w').mean()
```

Entrée [ ]:
```
# Faire des statistiques suivant les valeurs d'une colonne d'un dataframe
df['close'].resample('w').agg(['mean','std','min','max'])
```

Entrée [ ]:
```
# Faire une moyenne mobile simple sur 7 jours
df.loc['2019','close'].rolling(window=7).mean()
```

Entrée [ ]:
```
# Faire une moyenne mobile exponentielle
df.loc['2019','close'].ewm(alpha=0.5).mean()
```

Utiliser la fonction Transform

Entrée [ ]:
```
# Insérer dans le df initial des aggrégations de group by dans de nouvelles colonnes
total_price = orders.groupby('order_id').item_price.transform('sum')
# Pour chaque commande, on calcule la somme, et on réinjecte au niveau du détail de la commande
```

Utiliser les fonctions stack et unstack

Entrée [ ]:
```
# Stack : transformer toutes les colonnes en lignes : Intitulé de la colonne et sa valeur
df.stack().reset_index()
```

Entrée [ ]:
```
# Présenter plus simplement un groupby à plusieurs niveaux
titanic.groupby(['Sex', 'Pclass']).Survived.mean().unstack()
```

Faire des calculs de pivot

Entrée [ ]:
```
# Faire un calcul de table pivot sur un dataframe, sur 2 valeurs de lignes et 1 valeur de colonne
pd.pivot_table(df, values="D", index=["Col1","Col2"], columns=["Col3"])
```

Entrée [ ]:
```
titanic.pivot_table(index='Sex', columns='Pclass', values='Survived', aggfunc='mean')
# Même tableau avec les totaux (lignes et colonnes)
titanic.pivot_table(index='Sex', columns='Pclass', values='Survived', aggfunc='mean', margins=True)
```

Création de valeurs dummies

Entrée [ ]:
```
column_x_dummies = pd.get_dummies(df.column_x).iloc[:, 1:]
```