Enseignant :	Louis RAYNAL	**Établissement :**	ICES
Contact :	l-raynal@ices.fr	**Année :**	2024-2025
Public :	L1 Maths		
Ressources :	https://github.com/LouisRaynal/pythonData		

TP 2 : Premières analyses

Mémos pour Mac :

{ = Alt + (} = Alt +) | = Alt + ⇧ + L
[= Alt + ⇧ + ( ] = Alt + ⇧ +)

Selon la version du clavier Alt peut être à remplacer par Option

Au cas où vous avez un problème avec les fenêtres de Spyder :
View > Window Layouts > Spyder Default Layout

Pratique

Avant de commencer, si ce n'est pas déjà fait, ouvrir le navigateur Anaconda, puis Spyder.

Ouvrir un nouveau fichier *.py* dans Spyder, que vous sauvegarderez sur votre bureau avec comme nom *TP2.py*.

Ajouter en début de fichier les lignes d'import des librairies que nous utiliserons lors de ce TP, soit :

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as ss
```

Exécuter ces lignes de code.

Lors de ce TP, nous allons aborder différentes étapes clés pour l'analyse d'un jeu de données. Nous utiliserons des données portant sur des céréales afin d'illustrer ces étapes.

1. Premier jeu de données : céréales

Nous allons nous intéresser à un jeu de données stocké dans un fichier *.csv* nommé *cereales.csv*. Il s'agit de données pour 77 paquets de céréales vendus aux États-Unis. Chaque ligne du jeu de données correspond à un paquet de céréales, pour lequel des informations (ou variables) sont recueillies et stockées en colonnes. Ces variables sont les suivantes :

- `produit` : le nom des céréales
- `fabricant` : le fabricant, encodé de la manière ci-dessous
  - `A` = American Home Food Products
  - `G` = General Mills
  - `K` = Kelloggs
  - `N` = Nabisco
  - `P` = Post
  - `Q` = Quaker Oats
  - `R` = Ralston Purina
- `type` : le type de céréales soit `F` pour froides, `C` pour chaudes
- `calories` (nombre dans une portion)
- `protéines` (en grammes dans une portion)
- `lipides` (en grammes dans une portion)
- `sodium` (en milligrammes dans une portion)
- `fibres` (en grammes dans une portion)
- `glucides` (en grammes dans une portion)
- `sucres` (en grammes dans une portion)
- `potassium` (en milligrammes dans une portion)
- `vitamines` : vitamines et minéraux - 0, 25 ou 100, indiquant le pourcentage par rapport aux recommandations de la FDA (*U.S. Food and Drug Administration*)
- `étagère` : niveau de l'étagère où est positionné le produit dans les supermarchés (1, 2 ou 3 en partant du bas)
- `poids` : poids en ounces dans une portion (1 ounce $\simeq$ 28.35 grammes)
- `tasses` : nombre de tasses dans une portion (1 tasse $\simeq 23,66$ centilitres)
- `note` : la note du produit par des consommateurs, allant de 0 à 100

Exercices

1. Identifier la population dont est issu cet échantillon de 77 individus.
2. Identifier le type de chaque variable collectée (quantitative continue/discrète, qualitative nominale/ordinale).

Pratique

Télécharger le fichier *cereales.csv* sur votre bureau, depuis le dossier `TP2` à l'adresse `https://github.com/LouisRaynal/pythonData` (utiliser le navigateur web **Google Chrome**). Pour cela, cliquer sur le fichier *cereales.csv* puis sur le bouton `Raw`, ensuite faire un clic droit et `Enregistrer sous...`. Utiliser le nom cereales.csv lors de la sauvegarde du fichier.
(Le fichier se trouve également sur le Classroom dédié).

2. Inspection préliminaire du jeu de données

Avant d'importer des données dans Python, il est conseillé d'ouvrir le fichier contenant les données afin d'avoir un **premier aperçu des données** à analyser, et **répondre à quelques questions préliminaires** ci-dessous (lorsque possible). Les réponses à ces questions vont nous faciliter la prochaine étape qu'est le chargement des données.

Exercices

1. Ouvrir le fichier *cereales.csv* avec le logiciel de votre choix.
2. Est-ce que la première ligne est une entête (c'est-à-dire qu'elle contient le nom des variables) ?
3. Y a t il des lignes inutiles en bas du fichier ? Si oui, combien ? (Des lignes de commentaires du créateur du jeu de données par exemple)
4. Comment sont séparées les données ? (Virgules, tabulations, espaces...)
5. Y a t il des valeurs manquantes, c'est-à-dire des valeurs vides dans le jeu de données ?
6. Combien de lignes et colonnes sont contenues dans le jeu de données ?

3. Chargement des données

Les jeux de données peuvent être stockés de diverses manières, par exemple dans une base de données, ou dans un ou plusieurs fichiers avec différents formats possibles. Ici il s'agit d'un seul fichier au format *.csv* que nous souhaitons lire dans Python pour obtenir un DataFrame. Pour cela nous utiliserons la fonction `read_csv` de la librairie `pandas`.

En Python, la succession d'étapes classiques pour lire un fichier de données est la suivante :

1. Se placer dans le répertoire où se trouve le jeu de données
2. Lister les fichiers se trouvant dans ce répertoire
3. Lire le fichier de jeu de données dans Python
4. Vérifier que les données ont été correctement importées

Code 1 – Lecture d'un jeu de données depuis un fichier

```
# 1. Se placer dans le répertoire où se trouve le jeu de données (le bureau ici)
os.chdir('/Users/Guest/Desktop')
# puis vérifier que l'on est sur le bon répertoire
os.getcwd()

# 2. Lister les fichiers se trouvant dans ce répertoire
os.listdir()

# 3. Lire le fichier de jeu de données dans Python, ici sous la forme d'un DataFrame
fichier = 'cereales.csv'
df = pd.read_csv(fichier, header=0, skipfooter=0, sep=',')
```

Exercices

1. Vérifier que le jeu de données est bien importé, notamment en termes de nombre de lignes et nombre de colonnes. Pour cela, vous pouvez utiliser la méthode `shape` d'un DataFrame.
2. A quoi correspondent les paramètres `header`, `skipfooter` et `sep` de la fonction `read_csv` ?

Remarque

Par défaut, toutes les colonnes d'un DataFrame ne sont pas affichées. Vous pouvez changer le nombre maximal de colonnes affichées en affectant le nombre souhaité à la variable `pd.options.display.max_columns`.

Pour afficher toutes les colonnes de votre DataFrame, affecter la valeur `None`, soit :

```
pd.options.display.max_columns = None
```

4. Nettoyage des données

Avant d'analyser un jeu de données, il est nécessaire qu'il soit **propre**. En effet, un jeu de données est la plupart du temps issu de saisies manuelles, ce qui signifie de **potentielles erreurs de saisies**. Travailler sur un jeu de données non propre pourrait fausser les résultats de vos analyses.

D'une manière générale, nous allons passer par une succession d'étapes, qui nous permettront de **nettoyer notre jeu de données**. La plupart de ces étapes sont :

1. Supprimer les **doublons**
2. Détecter et traiter les **valeurs manquantes**
3. Détecter et traiter les **valeurs aberrantes**
4. **Renommer** des colonnes/lignes
5. **Transformer** le contenu de lignes/colonnes

La partie nettoyage d'un jeu de données se fait habituellement après sa lecture, mais il est tout à fait possible de s'apercevoir d'anomalies à "nettoyer" dans un jeu de données au cours de son analyse.

4.1. Les doublons

Pour diverses raisons (mauvaises manipulations, erreurs lors des saisies) il est possible que des lignes apparaissent en plusieurs exemplaires dans votre jeu de données. Il est nécessaire de supprimer ces **doublons** pour ne pas fausser vos analyses.

Pour ce faire, `pandas` contient les méthodes `duplicated` et `drop_duplicates` applicables sur des DataFrames. La première permet d'identifier les lignes en plusieurs exemplaires, la deuxième permet de les supprimer.

Code 2 – Identifier les doublons

```
df.duplicated()

# il est alors facile de compter le nombre de doublons
df.duplicated().sum()
```

Ce jeu de données contient bien des doublons à supprimer !

Remarque

> Notez que la recherche de doublons peut se faire sur un sous-ensemble de colonnes du DataFrame, en spécifiant à `duplicated` les colonnes d'intérêt dans une liste.
> Par exemple `df.duplicated(['fabricant'])`

Nous pouvons maintenant utiliser `drop_duplicates` pour supprimer les lignes doublons.

Code 3 – Supprimer les doublons

```
df.drop_duplicates(inplace=True)
```

Attention

> Le paramètre `inplace` peut s'utiliser dans la plupart des méthodes d'un DataFrame `pandas`. Il permet de directement modifier le DataFrame lorsqu'il est égal à `True`. Nous le rencontrerons souvent.

Exercices

1. Vérifier qu'il n'y a pas le même nom qui apparaît plusieurs fois dans la colonne `produit` du DataFrame.
2. Vérifier qu'il y a bien des valeurs identiques pour la variables `type`.

4.2. Les valeurs manquantes

Dans un jeu de données, il arrive que certaines informations soient manquantes, car elles sont inconnues et n'ont pas pu être observées, ou n'existent simplement pas pour certains individus. On parle alors de valeurs *null* ou *NA* (*Not Available*), ou plus généralement de **valeurs manquantes**.

Les valeurs manquantes sont habituellement représentées par `pandas` par une valeur en `NaN` (*Not a Number*), ou dans la version de base de Python par la valeur `None`. Ces valeurs (`NaN` ou `None`) peuvent être facilement détectées, on parle alors de **valeurs sentinelles** qui alertent sur ces valeurs manquantes.

Détection des valeurs manquantes

La méthode `isnull` d'un DataFrame permet d'identifier ses valeurs égales à `NaN` ou `None`. Appliquons-la à notre jeu de données.

Code 4 – Détection de valeurs manquantes

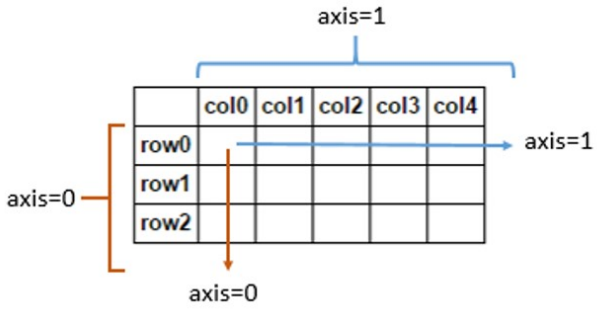
```
df.isnull()

# Comptons le nombre de fois où NaN est trouvé par colonne
df.isnull().sum(axis=0)
```

Remarque

Le paramètre `axis` peut s'utiliser dans la plupart des fonctions/méthodes de calcul de `pandas`. Elle permet d'indiquer la *direction* selon laquelle le calcul doit se faire.

- `axis=0` permet de faire le calcul sur toutes les lignes, par colonne donc.
- `axis=1` permet de faire le calcul sur toutes les colonnes, par ligne donc.



Une autre manière d'identifier rapidement les valeurs manquantes est d'utiliser la méthode `info` :

```
df.info()
```

Ce jeu de données n'a en apparence pas de valeurs manquantes. Cependant un jeu de données est habituellement accompagné d'une documentation, qui donne une description de son contenu, les colonnes qui le constituent et leurs unités si nécessaire, ainsi que de potentielles spécificités.

Une spécificité de ce jeu de données est que l'absence d'une information pour une paquet de céréales est encodée par la valeur `-1` ! Autrement dit : `-1` est une valeur sentinelle pour identifier les valeurs manquantes de ce jeu de données.

Nous souhaitons identifier s'il y a bien des colonnes où figure cette valeur sentinelle `-1`.

Code 5 – Détection de valeurs sentinelles (en -1)

```
df == -1

# Comptons le nombre d'occurences par colonne
(df == -1).sum(axis=0)
```

Astuce

Une méthode très utile est la méthode `any` d'un objet, qui renvoie `True` si l'une des valeurs de cet objet est `True`, sinon `False` est renvoyé. Cette méthode `any` accepte le paramètre `axis` précédemment expliqué, pour indiquer si la recherche doit se faire par ligne ou par colonne.

Cette fonction peut ici nous permettre d'afficher uniquement les lignes et colonnes qui contiennent ces fameuses valeurs `-1`. Voici comment procéder :

Code 6 – Extraire des lignes et colonnes avec conditions

```
(df == -1).any(axis=1) # renvoie True pour les lignes où on trouve une valeur -1

(df == -1).any(axis=0) # renvoie True pour les colonnes où on trouve une valeur -1

# Nous pouvons extraire de notre DataFrame les lignes et colonnes
# identifiées avec une valeur -1
df.loc[(df == -1).any(axis=1) , (df == -1).any(axis=0)]
```

On observe que les valeurs en `-1` se trouvent dans les colonnes `glucides`, `sucres` et `potassium`, aux lignes 4, 20 et 57. Le choix d'utiliser `-1` pour les valeurs manquantes n'est pas très malin car ces valeurs auraient pu affecter le calcul de statistiques descriptives. Si l'on avait par exemple calculé la moyenne des teneurs en potassium, cette moyenne serait biaisée par ces valeurs négatives !

Afin d'avoir un jeu de données correct, nous allons ré-importer notre jeu de données, mais cette fois en spécifiant que toutes les valeurs `-1` seront remplacées par des `NaN`. Pour ce faire, on utilise le paramètre `na_values` de la fonction `read_csv`.

Code 7 – Réimport des données céréales où -1 est un marqueur de valeur manquante

```
df = pd.read_csv(fichier, header=0, skipfooter=0, sep=',', na_values=[-1])
```

Notre jeu de données devrait maintenant contenir des valeurs manquantes !

Exercices

1. Étant donné que nous venons de réimporter le jeu de données, les doublons sont toujours présents. Recopier et exécuter à nouveau les étapes de suppression des doublons.
   La suite logique de votre script Python est de haut en bas.
2. Vérifier que vous avez bien des valeurs manquantes désignées par `NaN`.
3. En vous basant sur le code utilisé pour identifier les lignes et colonnes où se trouvaient les valeurs en `-1` (Code 6, dernière ligne), afficher les lignes et colonnes avec des valeurs manquantes.

Gestion des valeurs manquantes

Maintenant que nous avons identifié qu'il y a des valeurs manquantes, il faut décider de ce que l'on en fait. Il y a différentes manières de faire, les plus simples sont :

1. soit **supprimer** les lignes concernées ;
2. soit **remplacer** ces valeurs manquantes par d'autres valeurs.

Nous allons voir les deux méthodes, mais étant donné le peu de lignes que nous avons dans notre jeu de données, et pour éviter une perte d'individus, nous appliquerons la seconde.

1. La **suppression** des lignes avec valeurs manquantes peut se faire très facilement grâce à la méthode `dropna` d'un DataFrame.

```
df.dropna() # inplace=True aurait pu être ajouté pour appliquer la modification sur df
# Mais ne l'ajoutez pas ici, pour cet exemple on ne veut pas supprimer des lignes de df
```

Cette option est intéressante si les lignes concernées contiennent beaucoup de valeurs manquantes, c'est-à-dire dans notre cas certains paquets de céréales ont beaucoup d'informations non renseignées. Étant donné le peu d'informations manquantes, nous utiliserons la deuxième option.

2. Pour le **remplacement** de valeurs manquantes, l'idée est d'éviter la perte d'informations causée par la suppression de lignes. On souhaite plutôt remplacer les `NaN` (ou `None`) par des valeurs pertinentes. La méthode `fillna` est faite pour ça.

Donner une valeur constante à cette méthode remplacerait toutes les valeurs manquantes par cette constante. Par exemple

```
df.fillna(-1) # inplace=True aurait à nouveau pu être ajouté si vous vouliez modifier df
```

mais cela nous ramènerait à notre point de départ !

Une autre option est de lui donner un dictionnaire, où chaque clé correspond à un nom de colonne, et la valeur associée correspond à la valeur de remplacement des `NaN`. Une méthode consiste à remplacer les `NaN` par la **moyenne** ou la **médiane** des valeurs de chaque colonne.

Effectuons ici un remplacement par la valeur moyenne des colonnes `glucides`, `sucres` et `potassium`.

Code 8 – Remplacer les valeurs manquantes par les valeurs moyennes

```
1 dico = {'glucides':df.glucides.mean(),
2         'sucres':df.sucres.mean(),
3         'potassium':df.potassium.mean()}
```

```
# Veillez à exécuter les trois lignes ci-dessus en une seule fois
dico

df.fillna(dico, inplace=True)

df.loc[[4,20,57],['glucides','sucres','potassium']] # vérification
```

4.3. Les données aberrantes

D'une manière similaire aux données manquantes, il faut ouvrir l'œil pour détecter des **données aberrantes**, c'est-à-dire des données anormalement faibles ou élevées, ou même des valeurs impossibles au vu des variables recueillies.

Il existe différentes manières de repérer les données aberrantes, et la plupart du temps on ne les remarque pas tout de suite, mais plutôt lors de l'analyse des données. Par exemple, au moment de tracer un graphique, vous remarquez une donnée étrangement éloignée des autres, ou bien suite au calcul de statistiques descriptives telles que le minimum/maximum.

Que faire d'une donnée aberrante ? Cela dépend vraiment du contexte de votre donnée. Il faut se demander si cette valeur est une réelle erreur (de saisie par exemple), ou bien si c'est normal. Dans le premier cas il faudra la corriger si possible, dans le deuxième cas il faudra la laisser telle quelle.

Prenons l'exemple de notre jeu de données de céréales, toutes les valeurs numériques recueillies devraient être positives. Vérifions si c'est bien le cas. Pour ce faire, nous pouvons utiliser la méthode `describe` d'un DataFrame qui permet de calculer des statistiques descriptives, par défaut sur les variables numériques.

Code 9 – Calcul de statistiques descriptives

```
df.describe()
```

Remarquez que les colonnes `tasses` et `note` ont des valeurs négatives, on peut le voir sur les minimums calculés. Il s'agit de données aberrantes.

Le code ci-dessous permet d'identifier les valeurs négatives pour les colonnes numériques. Comprenez le code ci-dessous et exécutez-le ligne par ligne.

Code 10 – Identification de données aberrantes

```
df.select_dtypes(include=[np.number])

col_num = df.select_dtypes(include=[np.number]).columns

(df.loc[:,col_num] < 0).sum(axis=0) # combien de valeurs < 0 par colonne ?
(df.loc[:,col_num] < 0).sum(axis=1) # combien de valeurs < 0 par ligne ?

df[col_num].loc[ (df.loc[:,col_num]<0).any(axis=1) , (df.loc[:,col_num]<0).any(axis=0) ]
```

Exercices

Dans le code précédent, pourquoi doit-on extraire les colonnes numériques avant de faire la comparaison avec `0` ? Pour comprendre cela, essayez d'exécuter

```
df < 0
```

À vrai dire, j'ai ajouté intentionnellement ces valeurs négatives. Elles devraient être des valeurs positives. Il faut maintenant remettre ces valeurs en positif. Pour cela, une solution simple est d'appliquer la méthode `abs` qui permet de calculer la valeur absolue.

Code 11 – Correction de données aberrantes

```
df.loc[:,col_num] = df.loc[:,col_num].abs()
# Vérifiez qu'il n'y a plus de valeurs négatives
```

Astuce

> Notez que la ligne `df.select_dtypes(include=[np.number])` permet de sélectionner les colonnes de type numérique. D'une manière similaire, nous pouvons extraire les variables non-numériques avec `df.select_dtypes(include=[object])`.

Remarque

> Par chance, les colonnes concernées par ces données aberrantes ne sont pas les mêmes que pour les données manquantes que nous avons remplacées par la moyenne des colonnes. En effet, les moyennes n'ont pas été impactées. Dans le cas contraire, il aurait fallu corriger ces données aberrantes dans un premier temps, puis dans un deuxième temps remplacer les données manquantes.

4.4. Transformation de données

Pour finir sur le nettoyage des données, il y a diverses transformations/modifications du jeu de données qui peuvent être intéressantes pour faciliter son analyse.

Décodage de colonnes

Le descriptif des colonnes du jeu de données céréales nous indique qu'il y a deux variables qualitatives nominales : `fabricant` et `type`, qui utilisent des codes. Afin de faciliter la lecture du jeu de données, nous voudrions directement utiliser le label associé à un code. Par exemple, `froides` au lieu de `F` et `chaudes` au lieu de `C` pour la colonne `type`.

Pour réaliser de telles transformations, nous pouvons utiliser la méthode `map` d'une colonne d'un DataFrame. Cette méthode recevra un **dictionnaire** indiquant le codage de la colonne.

Code 12 – Décodage d'une colonne grâce à un dictionnaire

```
dicoType = {'F':'froides', 'C':'chaudes'}

df.type = df.type.map(dicoType)

df.type.unique() # vérification
```

Une alternative est de donner à `map` une **fonction** pour faire la traduction.

Code 13 – Décodage d'une colonne grâce à une fonction

```
def decodeType(x):
    if x == 'F':
        return 'froides'
    elif x == 'C':
        return 'chaudes'
```

```
    else:
        return x

df.type = df.type.map(decodeType)
```

La deuxième méthode est plus longue, mais peut être intéressante dans le cas où vous voulez créer des classes sur des colonnes numériques, par exemple.

Exercices

> Traduire la colonne `fabricant` afin d'afficher le nom complet des fabricants dans votre DataFrame.

Renommer des colonnes ou lignes

Il est aussi important d'avoir des noms de lignes et de colonnes qui vous conviennent et vous parlent. Un DataFrame possède la méthode `rename` pour cet usage. D'une manière similaire au décodage de colonnes, nous pouvons donner des fonctions ou bien des dictionnaires pour indiquer les transformations à effectuer.

Voici un exemple de son utilisation pour renommer deux lignes grâce à un dictionnaire, et mettre tous les noms de colonnes en majuscules grâce à la fonction `str.upper`.

Code 14 – Utilisation de la méthode `rename`

```
df.rename(index={0:'zero',1:'un'}, columns=str.upper)
```

Notez que la ligne ci-dessus ne modifiera pas `df`, mais retournera un autre DataFrame. (Pour s'en assurer, affichez `df`). A nouveau, c'est en ajoutant l'argument `inplace=True` que l'on aurait pu modifier `df`.

Exercices

> Remarquez que le nom de colonne `étagère` n'est pas bien orthographié. Utilisez la méthode `rename` pour corriger ce nom de colonne, et vérifiez que la modification a bien été faite.

Conversion de colonnes

Notez que certaines colonnes sont dans des unités peu utilisées en France : la colonne `poids` qui est en ounce, et la colonne `tasses` qui est en tasse.

Exercices

> Afin de rendre nos données plus parlantes, convertir les colonnes `poids` et `tasses`, respectivement en grammes et en centilitres (1 ounce $\simeq$ 28.35 grammes et 1 tasse $\simeq$ 23.66 centilitres).

Création/suppression de colonnes

Enfin, vous pouvez facilement créer de nouvelles colonnes, en faisant une affectation sur une colonne du DataFrame qui n'existe pas encore. La colonne sera ainsi créée.

Code 15 – Exemple de création d'une nouvelle colonne

```
df['Produit_maj'] = df.produit.str.upper()
df
```

Cela n'a pas trop d'intérêt ici, mais c'est plutôt à titre d'illustration.
Vous pouvez supprimer cette colonne en utilisant la méthode `drop` d'un DataFrame.

Code 16 – Exemple de suppression d'une colonne

```
df.drop(columns = 'Produit_maj', inplace=True)
```

(`drop` accepte aussi le paramètre `index` si vous voulez supprimer des lignes.)

5. Exploration/Analyse du jeu de données

Maintenant que notre jeu de données est nettoyé, nous pouvons enfin passer à son analyse.

Un jeu de données n'arrivera habituellement pas entre vos mains par hasard. Il y a des questions auxquelles vous voudrez répondre suivant les demandes ou buts d'une étude.

Concernant ce jeu de données, des exemples de questions pourraient être :

- Quelles sont les caractéristiques des différents fabricants en termes de qualité de céréales produites ?
- Quels sont les *pires* céréales/fabricants ?
- Quels sont les liens entre les différents apports nutritionnels ?

Quoi qu'il en soit, suite au nettoyage et à la préparation d'un jeu de données, il est bon d'**explorer votre jeu de données**, c'est-à-dire de mieux comprendre les données qui s'y trouvent. Cela se fait habituellement via la **visualisation** des données et le calcul de **statistiques descriptives**. Explorer un jeu de données vous ouvrira souvent de nouvelles pistes d'étude.

5.1. Visualisation de données

Créer des **visuels** informatifs est une étape importante dans l'analyse de données. Cela peut aussi faire partie du processus de nettoyage des données, par exemple pour aider à identifier des données aberrantes, ou bien pour la restitution d'informations auprès d'autres personnes.

Les visuels à adopter vont dépendre de la nature de vos variables. Nous allons voir comment tracer différents graphiques selon la nature des données. Les principales librairies Python pour effectuer des représentations de données sont `matplotlib` et `seaborn`. `pandas` propose aussi des fonctionnalités pour tracer des graphiques à partir de DataFrames, il s'agit en fait d'appels détournés à la librairie `matplotlib`.

Rappels sur la création de figures avec `matplotlib`

Nous rappelons ici comment tracer des figures avec la librairie `matplotlib`, et plus particulièrement avec son module `pyplot`. L'import de ce module se fait habituellement de la manière suivante :

```
import matplotlib.pyplot as plt
```

`matplotlib` utilise des objets de type `figure`, qu'il faudra compléter avec de nouvelles lignes de code pour obtenir des graphiques avec les bons labels, tailles des axes, ...

Voici quelques exemples pour tracer des courbes, histogrammes et nuages de points (**pas besoin de les réécrire**).

Code 17 – Exemples de courbes avec `matplotlib`

```
fig = plt.figure()

ax1 = fig.add_subplot(2, 1, 1)
ax1.plot(df.calories, linestyle = '', color='b', marker='.')
```

```
ax1.set_ylabel('Nombre de calories')
ax1.set_ylim([0,200])
ax1.set_title('Titre graphe du haut')

ax2 = fig.add_subplot(2, 1, 2)
ax2.plot(df.sucres, linestyle = ':', color='r')
ax2.set_ylabel('Sucres (mg)')
ax2.set_title('Titre graphe du bas')

plt.subplots_adjust(wspace=0, hspace=0.5)
```

Code 18 – Exemples d'un histogramme et d'un nuage de points avec `matplotlib`

```
fig = plt.figure()

ax1 = fig.add_subplot(2, 1, 1)
ax1.hist(df.calories, bins=10, color='r')

ax2 = fig.add_subplot(2, 1, 2)
ax2.scatter(df.sucres, df.calories)
ax2.set_xlabel('Sucres (mg)')
ax2.set_ylabel('Nombre de calories')

plt.subplots_adjust(wspace=0, hspace=0.25)
```

Le package `matplotlib` est en résumé assez bas niveau, dans le sens où il faut assembler la figure ligne par ligne en partant de presque rien. C'est pourquoi on privilégiera les possibilités offertes par les librairies `pandas` et `seaborn` plus faciles d'utilisation.

Création de figures avec `pandas` et `seaborn`

`pandas` propose des méthodes afin de faciliter la création de figures en se basant sur les données contenues dans un DataFrame. La librairie `seaborn` est une autre librairie de visualisation qui simplifiera la création de figures.

• **Courbes.** Pour les **variables quantitatives**, la méthode `plot.line` permet de tracer des courbes. Si aucun argument n'est donné, une courbe est tracée pour chaque colonne numérique du DataFrame (l'abscisse sera le nom des lignes). Allez dans l'onglet **Graphes** pour voir les graphiques tracés.

Code 19 – Exemples de courbes

```
df.plot.line()

df[['sucres','calories', 'potassium']].plot.line()

df[['sucres','calories', 'potassium']].plot.line(subplots=True)

df[['sucres','calories', 'potassium']].plot.line(figsize=(7,7), title='Mon titre ici')
```

Pour représenter une courbe en fonction de deux variables (quantitatives), il est aussi possible de spécifier les colonnes d'intérêt grâce aux paramètres `x` et `y`.

Code 20 – Exemples de courbes avec `plot.line`

```
df.plot.line(x='sucres', y='calories')
```

Les courbes se prêtent bien aux données ordonnées ou avec une temporalité (l'évolution d'une quantité à travers le temps par exemple). Ici, vu que nos données ne sont pas ordonnées, ça ne ressemble pas à grand-chose.

• **Diagrammes en bâtons.** Ce type de graphique permet de représenter une **donnée numérique en fonction d'une variable quantitative discrète ou qualitative**. Cela peut se faire simplement via la méthode `plot.bar` d'un DataFrame `pandas`.

L'exemple typique est de représenter la distribution des effectifs d'une variable qualitative, par exemple pour le `type` de céréales :

Code 21 – Exemple de diagrammes en bâtons

```
df.type.value_counts()
df.type.value_counts().plot.bar()
```

Exercices

> Réaliser le même genre de graphique pour représenter le nombre de céréales différentes produites selon les fabricants.

• **Camemberts.** Étant donné une **variable numérique**, un camembert peut facilement être tracé avec la méthode `plot.pie` d'un DataFrame.

Code 22 – Exemple de camemberts

```
df.type.value_counts().plot.pie(ylabel='', autopct='%.1f')
# autopct permet d'afficher le % calculé et indique la précision du calcul,
# ici un nombre flottant avec un chiffre après la virgule
```

Exercices

> Réaliser le même genre de graphique pour représenter la proportion de céréales produites selon les fabricants, et afficher les pourcentages avec deux chiffres après la virgule.

• **Histogrammes.** Représenter l'histogramme d'une **variable quantitative continue** permet de comprendre la distribution de ses valeurs rangées en classes. La méthode `plot.hist` permet de générer ce genre de représentations.

Code 23 – Exemples d'histogrammes

```
# En effectifs
df.calories.plot.hist(bins=10)

# En fréquences
df.calories.plot.hist(bins=10, density=True)
```

Le paramètre `bins` permet de définir le nombre de classes de l'histogramme, et `density` permet de spécifier si l'ordonnée doit être la **densité de fréquence** ou non. `density=True` est habituellement l'option à adopter car l'histogramme est alors un estimateur de la densité de la variable représentée.

Exercices

Tracer un histogramme des notes en densité de fréquence. Remarquez-vous quelque chose d'anormal ?

• **Nuages de points.** Un nuage de points permet de visualiser le lien entre **deux variables** (la plupart du temps **quantitatives**). Nous pouvons utiliser la méthode `plot.scatter` d'un DataFrame, avec les paramètres `x` et `y` pour indiquer les variables d'intérêt.

Code 24 – Exemple de nuages de points

```
df.plot.scatter(x='lipides', y='calories')
```

Exercices

1. Tracer des nuages de points montrant le lien entre le nombre de calories et la note. Que constatez-vous ?
2. Même question pour le lien entre la teneur en sucres et la note.

La librairie `seaborn` possède une fonction `pairplot`, permettant de tracer tous les nuages de points entre les paires de variables numériques possibles. Cela permet d'avoir un aperçu rapide des potentiels liens qui existent entre les variables.

Code 25 – Exemple de la fonction `pairplot` de `seaborn`

```
import seaborn as sns

sns.pairplot(df[['calories', 'protéines', 'lipides','sodium', 'note']])

sns.pairplot(df[['fibres', 'glucides', 'sucres', 'potassium', 'note']])
```

• **Boîtes à moustaches.** Une boîte à moustaches (ou boxplot) est un visuel qui permet de représenter la distribution d'une **variable quantitative continue**, notamment grâce aux quartiles (quantile à 25%, médiane et quantile à 75%) qui constitueront la "boîte" de ce type de graphe.

Par exemple, pour obtenir un boxplot sur les notes des céréales :

Code 26 – Exemple de boxplots

```
df.plot.box(column='note', rot=90)
```

On voit ainsi rapidement que 50% des notes sont comprises à peu près entre 35 et 50.

Remarque

Notez que les barres inférieures et supérieures ne représentent pas le minimum et le maximum, mais respectivement les bornes $Q_1 - 1.5 \times IQR$ et $Q_3 + 1.5 \times IQR$, où Q_1 et Q_3 désignent respectivement le premier et troisième quartile, et IQR désigne l'écart-interquartile $Q_3 - Q_1$. Les données en dehors de ces bornes sont représentées par des points et peuvent être de **potentielles données aberrantes**.

On remarquera par exemple que l'un des paquets de céréales a une note très élevée par rapport aux autres. A creuser plus tard !

L'un des avantages du boxplot est de représenter une **variable quantitative continue, en fonction d'une variable qualitative ou quantitative discrète**. Nous pouvons ajouter l'argument `by` suivi d'un nom de variable, pour obtenir un boxplot par modalité de cette dernière variable.

Code 27 – Exemple de boxplots par modalité

```
df.plot.box(column='note', by='étagère', rot=90)
```

On remarque que les céréales disposées sur la deuxième étagère ont de plus mauvaises notes.

Vous pouvez aussi spécifier plusieurs variables quantitatives continues dans le paramètre `columns`.

Code 28 – Exemple de boxplots par modalité

```
df.plot.box(column=['sucres','note'], by='étagère', rot=90)
```

On remarque que les céréales de la deuxième étagère ont aussi plus de sucres !

Exercices

> Réaliser un graphique permettant de visualiser la distribution des notes et de la teneur en fibres en fonction du fabricant de céréales. Y a-t-il un fabricant qui semble "spécialisé" dans les céréales à haute teneur en fibres ? Et y a-t-il un fabricant avec des notes particulièrement élevées ?

5.2. Résumés statistiques des données

En dehors de visuels parlants, il est aussi important de savoir caractériser et comprendre la distribution de vos données d'un point de vue numérique, grâce à des **résumés statistiques**.

Statistiques pour une variable

Il existe des résumés statistiques décrivant la **tendance centrale** d'une variable, tels que la **moyenne**, la **médiane**, le **mode**. Ils permettent de répondre à la question : si je devais résumer ma variable en une valeur, laquelle ce serait ?

Les DataFrames `pandas` contiennent de nombreuses méthodes permettant de calculer directement ou indirectement ces résumés statistiques, de même que la librairie `numpy`. Voici comment les calculer.

Code 29 – Calcul de statistiques de tendance centrale

```
# Moyenne
df.sucres.mean()  # avec pandas
np.mean(df.sucres) # avec numpy

# Médiane
df.sucres.median()
np.median(df.sucres)

# Mode
df.type.mode()
```

On trouve aussi des résumés statistiques décrivant la **dispersion des données**. Il s'agit par exemple de la **variance**, l'**écart-type**, l'**écart inter-quartile**, l'**étendue** (la différence entre maximum et minimum).

A nouveau, voici quelques manières de les calculer.

Code 30 – Calcul de statistiques de dispertion

```
# Variance
df.sucres.var(ddof=0)
np.var(df.sucres)
# Note : il existe une version de la variance avec N-1 au lieu de N au dénominateur.
# Par défaut pandas utilise N-1, lui préciser ddof=0 permet d'utiliser N.
# Par défaut numpy utilise N

# Ecart-type
df.sucres.std(ddof=0)
np.std(df.sucres)

# Ecart inter-quartile
# Calcul de quantiles (ici les 3 quartiles)
df.sucres.quantile(q=[0.25,0.5,0.75])
# Ecart inter-quartile :
df.sucres.quantile(q=0.75) - df.sucres.quantile(q=0.25)

# Etendue
np.ptp(df.sucres)
# vérification
df.sucres.max() - df.sucres.min()
```

Remarque

> Ces méthodes peuvent être utilisées sur les lignes ou colonnes d'un DataFrame, en précisant l'argument `axis=0` ou `axis=1`.

Enfin, nous avons déjà vu que la méthode `describe` d'un DataFrame permettait d'obtenir de nombreux résumés statistiques, selon la nature des variables.

Code 31 – Exemple d'utilisation de describe

```
df.describe(include=[np.number])

df.describe(include=[object])
```

Exercices

> Une règle de détection de données aberrantes consiste à identifier les données tombant en dehors de l'intervalle égal à $[Q_1 - 1.5 \times IQR\,;\, Q_3 + 1.5 \times IQR]$, où Q_1, Q_3 et IQR désignent respectivement le premier quartile, le troisième quartile et l'écart inter-quartile.
>
> Appliquer cette règle pour afficher les lignes des produits qui ont une note anormalement haute ou basse (et afficher toutes leurs colonnes).
> (On rappelle que l'opérateur logique "ou" entre deux conditions est `|` (`Alt+Maj+L` sur mac), et que vous pouvez utiliser la méthode `loc` sur votre DataFrame `df` pour en extraire des lignes et colonnes.)

Statistiques pour deux variables

On rappelle que la **covariance** entre deux variables quantitatives $\mathbf{x} = (x_1, \dots, x_N)$ et $\mathbf{y} = (y_1, \dots, y_N)$ est définie par

$$\mathrm{Cov}(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{\mathbf{x}})(y_i - \bar{\mathbf{y}})$$

où $\bar{\mathbf{x}}$ et $\bar{\mathbf{y}}$ désignent respectivement la moyenne de $\mathbf{x}$ et $\mathbf{y}$.

Une covariance positive indique que la relation entre $\mathbf{x}$ et $\mathbf{y}$ est (plus ou moins) croissante. Une covariance négative indique au contraire une relation décroissante.

Pour calculer la covariance entre deux variables quantitatives, nous pouvons utiliser la méthode `cov` d'un DataFrame `pandas`.

Code 32 – Calculs de covariances

```
df.calories.cov(df.note)
# La tendance semble décroissante

# Vérifions cela graphiquement
df.plot.scatter(x='calories', y='note')
```

Remarque

Il est possible de calculer la covariance entre toutes les variables numériques d'un DataFrame de la manière suivante :

```
df.cov()
```

Cela permet de rapidement identifier des potentielles relations fortes entre des paires de variables.

La **corrélation** entre deux variables $\mathbf{x}$ et $\mathbf{y}$ se calcule de la manière suivante :

$$\mathrm{Corr}(\mathbf{x}, \mathbf{y}) = \frac{\mathrm{Cov}(\mathbf{x}, \mathbf{y})}{\sqrt{\mathrm{Var}(\mathbf{x})}\sqrt{\mathrm{Var}(\mathbf{y})}}$$

Elle mesure le lien **linéaire** entre ces deux variables. Une valeur de 1 ou -1 indique que le nuage de points entre les deux variables est une droite, et 0 signifie qu'il ne présente aucune relation linéaire.

La méthode `corr` d'un DataFrame s'utilise de manière similaire à la covariance.

Code 33 – Calculs de correlations

```
df.calories.corr(df.note)

df.corr()
```

Exercices

Quelle est la variable qui semble présenter la relation la plus linéaire avec la variable `note` ? (autre que `note` elle-même)
Représenter graphiquement cette relation.

6. Analyse par groupe

L'analyse de données s'intéresse souvent à des **groupes d'individus**. Créer ces groupes et caractériser chacun d'entre eux est une étape importante de l'analyse de données. Ces groupes peuvent soit être des modalités de variables qualitatives (la variable `fabricant` par exemple), soit des valeurs de variables quantitatives avec peu de valeurs (la variable `étagère` par exemple), ou bien des groupes que vous avez créés vous-même.

Regrouper un DataFrame

`pandas` propose une méthode `groupby` permettant de travailler sur des groupes. Cette méthode renvoie un objet de type `DataFrameGroupBy`, c'est-à-dire un DataFrame groupé.

Grouper notre DataFrame selon le fabricant est très simple :

Code 34 – Utilisation de `groupby`

```
df.groupby('fabricant')
```

Pour chaque groupe, nous pouvons alors appliquer des méthodes de notre choix. En voici quelques exemples :

Code 35 – Calculs sur des groupes

```
# Pour chaque fabricant, la moyenne est calculée pour toutes les colonnes numériques de df
df.groupby('fabricant').mean()

# Pour chaque fabricant, le nombre d'individus du groupe est calculée
df.groupby('fabricant').size()
```

Des groupes peuvent même être créés selon des modalités jointes de plusieurs variables. Par exemple les modalités jointes des colonnes `fabricant` et de `type`.

Code 36 – Calculs sur des groupes conjoints

```
df.groupby(['fabricant','type']).size()
```

Si vous voulez limiter le calcul de résumés statistiques à certaines colonnes, vous pouvez filtrer les colonnes d'intérêt après l'appel à `groupby`. Par exemple :

Code 37 – Calculs sur des groupes avec restriction de colonnes

```
df.groupby('fabricant')['calories','sucres','note'].mean()
```

De plus, vous n'êtes pas limités à un seul calcul par groupe. Appliquer plusieurs fonctions à la fois est possible via la méthode `agg` qui recevra la liste des fonctions à appliquer sur chaque groupe.

Code 38 – Calculs multiples sur des groupes

```
df.groupby('fabricant').agg(func=[np.mean, np.median])

# Vous pouvez vous limiter à certaines colonnes, ici calories et sucres
df.groupby('fabricant')['calories','sucres'].agg([np.mean, np.median])

```

```
# Vous pouvez donner les noms de sortie que vous voulez
df.groupby('fabricant')['calories','sucres'].agg([('Moyenne',np.mean),('Médiane',np.median)])

# Vous n'êtes pas limités aux fonction de numpy.
# Pour utiliser des fonctions de pandas, mettre son nom entre ' ', par exemple :
df.groupby('fabricant').agg(func=['mean', 'count'])
```

Enfin, pour gagner plus de flexibilité dans vos calculs et ne pas vous limiter aux fonctions des librairies, vous pouvez appliquer sur chaque groupe des fonctions que vous aurez écrites. Pour cela, il faut utiliser la méthode `apply`.

Code 39 – Calculs personnalisés sur des groupes

```
# Créons une fonction qui renvoie les n premières lignes d'un DataFrame avec
# les valeurs les plus élevées d'une colonne
def nPremiers(monDataFrame, n=3, colonne='note'):
    return monDataFrame.sort_values(by=colonne, ascending=False)[:n]

# Vous pouvez appliquer cette fonction sur tout le DataFrame
nPremiers(df, n=2, colonne='note')

# Ou sur chaque groupe de fabricant !
df.groupby('fabricant').apply(nPremiers, n=2, colonne='note')
# On récupère ainsi pour chaque fabricant, ses 2 meilleurs paquets de céréales
```

Exercices

> Grouper votre DataFrame selon les modalités de la colonne `étagères`, puis calculer pour chaque modalité la moyenne et l'écart-type sur les colonnes `calories`, `protéines`, `sucres`, `vitamines`, `potassium` et `note`.

Créer des groupes personnalisés

Au lieu de vous limiter à des groupes basés sur des modalités de variables existantes, vous pouvez créer vos propres groupes. Par exemple, si vous voulez partitionner une variable quantitative continue en classes de valeurs, vous pouvez utiliser la fonction `cut`.

Créons ici 4 classes de notes basées sur les intervalles $[0;25]$, $]25;50]$, $]50;75]$, $]75;100]$.

Code 40 – Création de groupes personnalisés

```
groupNotes = pd.cut(df.note, bins=[0,25,50,75,100], include_lowest=True)
# bins reçoit les bornes des classes
groupNotes # On obtient un objet de type category

# On peut déterminer le nombre d'éléments dans chaque classe
groupNotes.value_counts()

# On peut aussi définir nos propres labels qui seront utilisés comme noms de classes
groupNotes = pd.cut(df.note, bins=[0,25,50,75,100], include_lowest=True, labels=['mauvais','moyen','bon','excellent'])

```

```
11 groupNotes.value_counts()
```

Une fois ces groupes créés, nous pouvons les utiliser dans une méthode `groupby`.

Code 41 – Calculs sur des groupes personnalisés

```
1 df.groupby(groupNotes).mean()
```

Exercices

La fonction `pd.qcut` permet de créer des intervalles qui répartissent les données de manière égale dans ces intervalles. Par exemple `pd.qcut(df.note, n)` créera `n` intervalles de notes de cette manière.

1. Utiliser cette fonction pour partitionner de manière égale les notes en 4 intervalles.
2. Vérifier que chaque intervalle contient bien (à peu près) le même nombre de notes.
3. Calculer pour chaque intervalle de note, la moyenne, la médiane et l'écart-type des teneurs en sucres, calories et lipides (en une seule ligne de code).

7. Distribution de variables

Vous avez vu lors du cours de statistiques descriptives, que la distribution d'une ou plusieurs variables pouvait être en effectifs ou en fréquences. Nous allons ici voir comment `pandas` permet de facilement calculer ces distributions.

7.1. Distribution d'une variable

Pour obtenir les effectifs d'une variable d'un DataFrame, nous avons déjà rencontré la méthode `value_counts`. La distribution en fréquences peut s'obtenir grâce à l'argument `normalize=True`.

Code 42 – Calcul des effectifs et fréquences d'une variable

```
1 # En effectifs
2 df.fabricant.value_counts()
3
4 # En fréquences
5 df.fabricant.value_counts(normalize=True)
```

Exercices

Pour les variables quantitatives continues, il est nécessaire de créer des classes à partir d'intervalles de valeurs. C'est ce que fait l'histogramme.

1. Choisir une variable continue du jeu de données (autre que note) et créer une partition de ses valeurs selon des classes de votre choix.
2. Pour ces classes, calculer les effectifs et les fréquences.

7.2. Distribution de deux variables

Pour calculer la distribution de **deux variables qualitatives** (ou quantitatives avec peu de valeurs différentes), nous pouvons utiliser la fonction `crosstab` de `pandas`.

Calculons par exemple la **distribution jointe** en effectifs, des variables `fabricant` et `type`.

Code 43 – Calcul de distribution jointe en effectifs

```
tabEffFabType = pd.crosstab(df.fabricant, df.type, margins=True)
tabEffFabType
```

On obtient alors un tableau de contingence.

Notez que l'argument `margins=True` permet d'obtenir les **distributions marginales**.

Exercices

- Calculer la distribution jointe des variables `fabricant` et `type` en **fréquences**.

Les **distributions conditionnelles** en fréquences peuvent simplement s'obtenir en jouant sur le paramètre `normalize`. En effet, `normalize='columns'` permet d'obtenir les distributions conditionnellement à chaque valeur en colonne. `normalize='index'` permet d'obtenir les distributions conditionnellement à chaque valeur en ligne.

Exercices

- Calculer les distributions conditionnelles de `fabricant` | `type`, puis `type` | `fabricant`.

8. Indépendance entre deux variables qualitatives

Grâce au cours de statistiques descriptives vous connaissez deux manières de mesurer l'**indépendance** entre **deux variables qualitatives** :

1. Par une **comparaison des distributions conditionnelles en fréquences**.
2. Par le **calcul de la statistique du χ^2**.

Notre but est ici de tester l'indépendance entre les variables `fabricant` et `type` de céréales. Nous allons voir ces deux méthodes en Python.

1. Comparaison des distributions conditionnelles

L'idée est de visualiser si la distribution de `fabricant` est influencée par les modalités de `type`. En d'autres termes, si les deux distributions conditionnelles $\texttt{fabricant}_{|\texttt{type=chaudes}}$ et $\texttt{fabricant}_{|\texttt{type=froides}}$ sont très proches (en fréquences) de la distribution marginale de `fabricant` (en fréquences), nous pourrons penser que les variables `fabricant` et `type` sont indépendantes. Si elles différent trop, on ne pourra pas conclure que les deux variables sont indépendantes.

Nous avons vu que les distributions conditionnelles des fabricants selon le type de céréales pouvaient s'obtenir facilement :

Code 44 – Distributions conditionnelles de `fabricant` sachant `type`

```
distFabCondType = pd.crosstab(df.fabricant, df.type, margins=True, normalize='columns')
distFabCondType

distFabCondType.plot.bar()
```

On constate que la distribution conditionnelle $\texttt{fabricant}_{|\texttt{type=chaudes}}$ diffère beaucoup des deux autres. On peut donc penser que `fabricant` et `type` ne sont pas indépendantes.

2. Calcul de la statistique du χ^2

On rappelle que le calcul de la statistique du χ^2 mesure l'écart à l'indépendance entre deux variables qualitatives. Elle découle de la comparaison entre les effectifs théoriques d'indépendance et les effectifs observés (revoir votre cours de statistiques descriptives si besoin).

- Une statistique du χ^2 proche de zéro signifiera que nos deux variables d'intérêt sont indépendantes.
- Au contraire, plus la statistiques du χ^2 est élevée, et plus on conclura à la non indépendance entre nos deux variables d'intérêt.

Pour effectuer un test d'indépendance de deux variables via la statistique du χ^2, nous pouvons utiliser la fonction `chi2_contingency` de la librairie `scipy.stats`. Il suffit de lui donner le tableau de contingence en effectifs entre nos deux variables, et elle nous renvoie plusieurs valeurs, notamment la statistique du χ^2 calculée.

Code 45 – Test d'indépendance du χ^2

```
import scipy.stats as ss

tabEff = pd.crosstab(df.fabricant, df.type)
tabEff

stat_chi2, p_value, dof, expected = ss.chi2_contingency(tabEff)
print(stat_chi2)
print(p_value)
```

On remarque ici que la statistique du χ^2 est très éloignée de 0. On ne peut donc pas dire que les deux variables sont indépendantes.

Pratique

> **Important :** Une autre quantité qui est utilisée dans les tests est la p-valeur (`p_value`).
> Pour l'instant, **retenez que si la p-valeur est inférieure à 5%, alors on rejette l'hypothèse que les deux variables sont indépendantes.**

La p-valeur est ici bien inférieure à 0.05, on rejette donc l'hypothèse selon laquelle les variables `fabricant` et `type` sont indépendantes.

9. Mesure de la relation entre une variable quantitative et une variable qualitative

Nous allons à nouveau revoir des notions vues dans le cours de statistiques descriptives que vous avez suivi. Le but étant de comprendre si une variable qualitative a un impact sur une variable quantitative.

Nous allons par exemple essayer de comprendre si la variable quantitative `note` est dépendante de la variable qualitative `fabricant`.

Idée 1 : Représenter les distributions conditionnelles

Nous pouvons représenter les boxplots des notes, par fabricant puis tous fabricants confondus, et observer s'il y a une différence notable.

Code 46 – Comparaison des boxplots de `note`, par `fabricant` ou non

```
# Lancer les trois lignes ci-dessous en même temps
fig, axes = plt.subplots(nrows=1, ncols=2)
df.plot.box(column='note', by='fabricant', rot=90, ax=axes[0])
df.plot.box(column='note', ax=axes[1])
```

Excepté pour le fabricant Nabisco, on n'observe pas de différence notable entre les boxplots par fabricant ou tous fabricants confondus.

Idée 2 : Comparer les moyennes et variances (conditionnelles contre marginale)

Au travers de résumés statistiques, nous pouvons comparer la tendance centrale et la dispersion des notes, selon les fabricants puis tous fabricants confondus.

Code 47 – Comparaison de la moyenne et variance de `note`, par `fabricant` ou non

```
df.groupby('fabricant')['note'].mean()
df.groupby('fabricant')['note'].var(ddof=0)
# A comparer à la version non conditionnelle
df['note'].mean()
df['note'].var(ddof=0)
```

On observe que le fabricant General Mills a une moyenne de notes bien en dessous de la note moyenne tous fabricants confondus. Le fabricant Nabisco semble à nouveau meilleur que les autres en termes de note moyenne.

Concernant les variances, on constate que Quaker Oats produit des céréales avec la plus grande variabilité de notes, alors que Nabisco et Ralston Purina ont une très faible variance de notes, en comparaison avec la variance calculée sur toutes les notes.

Idée 3 : Calcul du coefficient de détermination R^2

On rappelle que le coefficient de détermination R^2 mesure l'influence d'une variable qualitative $\mathbf{x}$ (ici le `fabricant`) sur une variable quantitative $\mathbf{y}$ (ici la `note`).

Le R^2 est issu de la décomposition de la variance totale en variance intra-groupes et variance inter-groupes. C'est-à-dire lorsque $\mathbf{x}$ possède K groupes, nous avons

$$\text{Var}(\mathbf{y}) = \underbrace{\frac{1}{N}\sum_{k=1}^{K} N_k \times \text{Var}(\mathbf{y}_k)}_{\text{variance intra-groupes}} + \underbrace{\frac{1}{N}\sum_{k=1}^{K} N_k \times (\bar{\mathbf{y}}_k - \bar{\mathbf{y}})^2}_{\text{variance inter-groupes}}$$

où $\text{Var}(\mathbf{y}_k)$ et $\bar{\mathbf{y}}_k$ désignent respectivement la variance et la moyenne de $\mathbf{y}$ pour le k-ième groupe, et N_k désigne l'effectif de ce groupe.

Le coefficient de détermination R^2 est égal à :

$$\begin{aligned} R^2 &= \frac{\text{variance inter-groupes}}{\text{variance totale}} \\ &= 1 - \frac{\text{variance intra-groupes}}{\text{variance totale}} \end{aligned}$$

Interprétations :
- Un R^2 égal à 1 signifie que la connaissance de $\mathbf{y}$ est entièrement déterminée par la connaissance de $\mathbf{x}$, c'est-à-dire dans notre cas que la note est entièrement déterminée par la connaissance du fabricant.
(*connaître le fabricant suffit à connaître la note*)
- Un R^2 égal à 0 signifie que la connaissance de $\mathbf{x}$ n'apporte strictement aucune information sur $\mathbf{y}$,
c'est-à-dire dans notre cas que connaître le fabricant n'aide pas à connaître la note.
(*la note n'est pas influencée par le fabricant*)

Exercices

1. Calculer la variance intra-groupe (bien utiliser la variance avec comme argument `ddof=0`).
2. Calculer la variance totale.
3. Calculer le coefficient de détermination R^2 qui devrait être égal à 0.36835.

Étant donné la valeur du coefficient de détermination, on peut ici dire que la note est modérément impactée par le fabricant.

10. Régression linéaire

On rappelle que la **régression** consiste à trouver une fonction $f : \mathbb{R} \to \mathbb{R}$ telle que $\mathbf{y} = f(\mathbf{x}) + \epsilon$, où $\epsilon : \mathbb{R} \to \mathbb{R}$ est un terme d'erreur qui sera aussi petit que possible.

Nous nous intéressons ici à la **régression linéaire**, où $f(\mathbf{x}) = a\mathbf{x} + b$, c'est-à-dire que la relation entre $\mathbf{y} = (y_1, \ldots, y_N)$ et $\mathbf{x} = (x_1, \ldots, x_N)$ est linéaire avec

$$\mathbf{y} = a\mathbf{x} + b + \epsilon$$

où a **et** b sont **deux paramètres à estimer**.

Ces deux paramètres sont estimés par les valeurs $\hat{a}$ et $\hat{b}$ qui minimisent la somme des erreurs quadratiques :

$$\sum_{i=1}^{N} (y_i - (a \times x_i + b))^2 \tag{1}$$

De nombreuses librairies Python implémentent la régression linéaire. Nous utiliserons la fonction `linregress` de la librairie `scipy.stats`.

Vous avez dû remarquer que la `note` et la teneur en `sucres` présentaient une relation plutôt linéaire (soit graphiquement ou parce que la corrélation entre ces deux variables est égale à -0.762181).

```
df.plot.scatter(x='sucres', y='note')
```

Ajustons une régression linéaire pour expliquer la `note` en fonction de la teneur en `sucres` d'un paquet de céréales, soit :

$$\texttt{note} = a \times \texttt{sucres} + b + \epsilon$$

Code 48 – Ajustement d'une regréssion linéaire

```
res = ss.linregress(df.sucres, df.note)
a_hat = res.slope     # il s'agit du a estimé
a_hat
b_hat = res.intercept # il s'agit du b estimé
b_hat
```

Nous venons de déterminer les valeurs optimales pour a et b, dénotées $\hat{a}$ et $\hat{b}$.

Étant donné de nouvelles teneurs en sucres pour de nouvelles céréales, nous pouvons maintenant prédire la note de ces céréales !

Code 49 – Prédictions

```
val_sucres = np.array([0,20]) # pour des teneurs en sucres égales à 0 et 20
val_notes = a_hat * val_sucres + b_hat
val_notes # voici les notes prédites
```

Nous pouvons aussi visualiser la courbe que nous avons ajusté, en prenant deux points assez éloignés sur cette courbe (ceux que nous venons de calculer par exemple !)

Code 50 – Visualisation de la droite de régression linéaire $\texttt{note} = \hat{a} \times \texttt{sucres} + \hat{b}$

```
# Lancer les deux lignes ci-dessous en même temps
df.plot.scatter(x='sucres', y='note')
plt.plot(val_sucres, val_notes)
```

Exercices

1. Ajuster le modèle de régression linéaire entre la `note` et le nombre de `calories`.
2. Selon ce modèle linéaire, quelle est la note qu'aurait un paquet de céréales avec un nombre de calories de 40 ?
3. Même question pour 180 calories.
4. Sur le même graphique, représenter la droite de régression linéaire ainsi que le nuage de points entre les variables `calories` et `note`.