Statistique : Premières analyses avec le logiciel Python

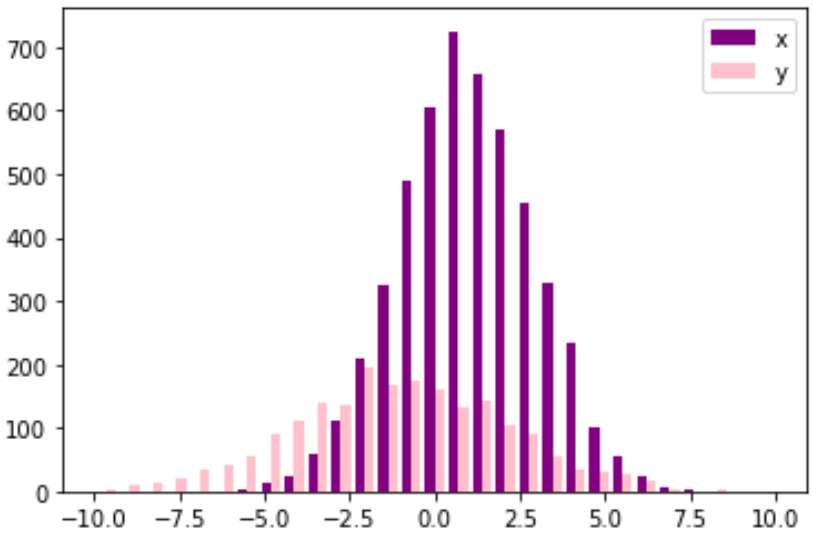
Christophe Chesneau

Université de Caen

Statistique : Premières analyses avec le logiciel Python

Christophe Chesneau

https://chesneau.users.lmno.cnrs.fr/



Caen, le 26 Juin 2022

Table des matières

1 Introduction **7**
1.1 Spyder . . . 7
1.2 Premiers pas . . . 8
1.3 Tableau de données . . . 12
1.4 Manipulations d'un tableau de données . . . 13
1.5 Importation de données . . . 15

2 Lois de probabilité **17**
2.1 Loi normale . . . 17
2.1.1 Densité . . . 17
2.1.2 Fonction de répartition . . . 19
2.1.3 Fonction de quantile . . . 22
2.1.4 Generation de valeurs . . . 22
2.1.5 Mesures de moments . . . 24
2.1.6 Complements . . . 26
2.2 Loi de Poisson . . . 26
2.2.1 Probabilité de masse . . . 26
2.2.2 Fonction de répartition . . . 28
2.2.3 Fonction de quantile . . . 28
2.2.4 Generation de valeurs . . . 29
2.2.5 Mesures de moments . . . 30
2.2.6 Complements . . . 31

3 Données de référence : `tips` **33**

4 Statistique descriptive **35**
4.1 Description de caractères quantitatifs . . . 35
4.2 Description de caractères qualitatifs . . . 38
4.3 Description conjointe de caractères quantitatifs . . . 40
4.4 Description conjointe de caractères qualitatifs . . . 41
4.5 Description conjointe d'un caractère quantitatif et d'un caractère qualitatif . . . 43
4.6 Description conjointe de deux caractères quantitatifs et d'un caractère qualitatif . . . 45

5 Tests statistiques **47**
5.1 Test de la normalité . . . 47
5.2 Test d'une moyenne . . . 50

5.3 Test de comparaison de deux moyennes (échantillons indépendants) 51
5.4 Test d'indépendance de deux caractères quantitatifs 52
5.5 Test de corrélation 53
5.6 Test d'une proportion 54
5.7 Test de comparaison de deux proportions (échantillons indépendants) 54
5.8 Test de positionnement 55
5.9 Test de comparaison de plusieurs moyennes (ANOVA) 56
5.10 Intervalles de confiance 57
5.10.1 Intervalle de confiance pour une moyenne 57
5.10.2 Intervalle de confiance pour une proportion 58

6 Élements de classification et de régression 61
6.1 Classification 61
6.2 Régression linéaire 62
6.2.1 Régression linéaire simple 62
6.2.2 Régression linéaire multiple 65
6.3 Régression logistique simple 66

~ Note ~

Ce document propose une introduction à l'analyse de données avec le logiciel Python.

On y aborde principalement les opérations élémentaires, l'utilisation des lois de probabilité, la statistique descriptive, les tests statistiques, et quelques éléments de classification et de régression (linéaire et logistique).

Ce document vient compléter le polycopié du cours de Statistique fondamentale 1 du Master Statistique de l'université de Caen, qui traite exclusivement du logiciel R. Il est donc supposé que les notions abordées sont connues un minimum (les formules sous-jacentes et les subtilités sont parfois omises).

N'hésitez pas à me contacter pour tout commentaire :

`christophe.chesneau@gmail.com`

Bonne lecture !

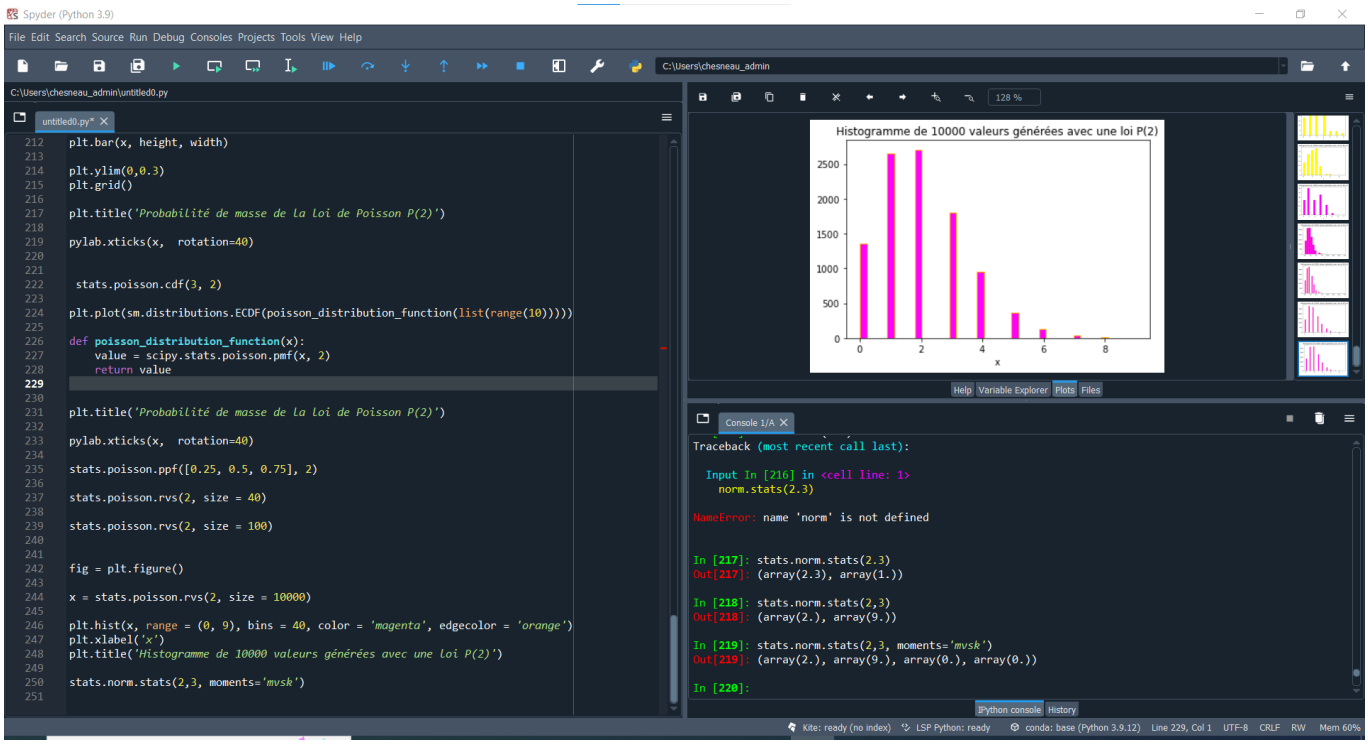
1 Introduction

1.1 Spyder

Avant toute chose, on recommande d'installer la distribution Python appelée **Anaconda** disponible ici :

`https://www.anaconda.com/products/distribution`

Il est conseillé de travailler avec l'interface **Spyder**. Vous allez avoir un espace de travail qui se présente de la manière suivante :



Dans la grande fenêtre de gauche, on tape les codes, comme un fichier texte quelconque. On les exécute en sélectionnant les lignes nécessaires, puis en tapant `F9`. Les résultats numériques s'affichent dans la fenêtre en bas à droite. Si graphique il y a, on peut le visualiser en haut à droite, ainsi que toutes les variables définies, et autres.

Pour commencer, dans la fenêtre de saisie, on importe certains outils appelés modules qui vont nous permettre de faire des statistiques et des graphiques. Ainsi, dans la fenêtre de saisie, on écrit :

```
import matplotlib.pyplot as plt
import scipy.stats
```

```
import numpy as np
import pylab
import pandas as pd
import seaborn as sns
import statistics
import statsmodels.api as smi
```

puis on exécute ces commandes (avec `F9` après les avoir toutes sélectionnées, ou en appuyant plusieurs fois sur `F9`, ou autre). **On les importe une fois pour toute dans tout le document.** De brèves présentations de ces modules sont données ci-dessous :

- `matplotlib` est un module offrant une bibliothèque complète pour créer des visualisations statiques, animées et interactives.
- `scipy` est un module offrant un grand nombre de lois de probabilité, et d'outils statistiques.
- `numpy` est un module offrant des fonctions mathématiques complètes, des générateurs de nombres aléatoires, et diverses routines d'algèbre linéaire.
- `pylab` est un module permettant d'utiliser de manière aisée les modules NumPy et matplotlib.
- `pandas` est un module offrant des structures de données adaptées à l'analyse statistique et des fonctions facilitant l'accès aux données, l'organisation des données et la manipulation des données.
- `seaborn` est un module offrant une interface de haut niveau et concise pour l'obtention de graphiques statistiques informatifs et attractifs.
- `statistics` est un module offrant des fonctions pour calculer des statistiques mathématiques de données numériques (à valeur réelle).
- `statsmodels` est un module offrant des classes et des fonctions pour l'estimation de divers modèles statistiques, pour les tests statistiques et pour l'exploration des données.

Pour une description détaillée de ces modules, on peut faire, par exemple : `help(scipy)`

1.2 Premiers pas

Ici, on commence à utiliser Python avec des opérations simples sous la forme d'exemples choisis. Le but étant de comprendre les grandes lignes de la syntaxe de Python.

- Dans la fenêtre de saisie (on ne le dira plus désormais), on fait :

  ```
  5 + 3 # cela va faire 8
  ```

  On exécute (avec `F9` ou autre, on ne le dira plus désormais), et cela renvoie : `8`

  Ainsi, on peut utiliser Python comme une simple calculatrice, et mettre des commentaires sur ce que l'on veut à côté grâce à `#`.
- On fait :

  ```
  5 * 3, 5 ** 3
  ```

  Cela renvoie : `(15, 125)`. Ainsi, la multiplication se fait avec $*$, la puissance avec $**$ et la

virgule permet de mettre plusieurs opérations côte à côte. Une commande équivalente à `5 ** 3` est `pow(5, 3)`.

— On fait :

```
a, b, c =  3, 5, 7
```

Dès lors, cela revient à affecter à `a` la valeur 3, à `b` la valeur 5, et à `c` la valeur 7.

— Ensuite, on fait :

```
a / b, a - b / c, (a - b) / c
```

Cela renvoie : `(0.6, 2.2857142857142856, -0.2857142857142857)`

— On fait :

```
print("la valeur de", a, "+", b, "est :", a + b)
```

Cela renvoie : `la valeur de 3 + 5 est : 8`

— On fait :

```
np.array([a, b, c])
```

Cela renvoie : `array([3, 5, 7])`. Ainsi, on a créé le vecteur numérique (a, b, c), correspondant à $(3, 5, 7)$.

— On fait :

```
np.sqrt(c + b - a) == 3
```

Cela renvoie : `True`. Ainsi, on a appelé la fonction "racine carrée" dans `numpy` (abregé en `np`) pour finalement demander si $\sqrt{9}$ est égale à 3, d'où le `True`. En fait, les opérateurs de comparaisons disponibles dans Python sont : `==` qui signifie " égal à", `!=` qui signifie "différent de", `>` qui signifie "strictement supérieur à", `>=` qui signifie "supérieur ou égal à", `<` qui signifie "strictement inférieur à" et '`<=` qui signifie "inférieur ou égal à".

— On fait :

```
type(np.sqrt(c + b)), type("test")
```

Cela renvoie : `(numpy.float64, str)`. Ainsi, la commande `type` nous permet de savoir de quel type est l'objet considéré. Ici, `float64` signifie que c'est un nombre réel et `str` que c'est du texte.

— On fait :

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [list1, list2]
List3
```

Cela renvoie : `[[1, 2, 3], [4, 5, 6]]`. On a ainsi créé une liste de valeurs, constituée de deux sous-listes.

— On fait :

```
list1[0] + list3[1][0]
```

Cela renvoie : `5`. Ainsi, on a cherché le premier élément de `list1` (Python commençant à l'indice 0, contrairement à R notamment), donc 1, et le premier élément de la deuxième sous-liste de

`list3`, soit `list2`, donc 4, et on les a additionné, ce qui donne 5.

— On fait :

```
tab1 = np.array([0, 1, 2])
tab2 = np.array([3, 4, 5])
tab3 = np.array([6, 7, 8])
mat1 = np.array([tab1, tab2, tab3])
```

Cela renvoie :

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Ainsi, on a créé trois vecteurs numériques pour former une matrice de dimension 3×3 nommée `mat1`.

— On fait :

```
mat1[1][2]
```

Cela renvoie : `5`. Ainsi, on a demandé la composante de cette matrice située à l'intersection de la deuxième colonne et troisième ligne, soit 5.

— On fait :

```
mat2 = np.reshape(np.arange(9), (3,3))
mat2 == mat1
```

Cela renvoie :

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Ainsi, on a construit une matrice `mat2` formée des entiers allant de 0 à 8, avec 3 lignes et 3 colonnes, ce qui correspond exactement à `mat1`, d'où la sortie.

— On fait :

```
mat1[1][2] = 6,
np.linalg.inv(mat1)
```

Cela renvoie :

```
array([[-1.66666667,  1.        , -0.33333333],
       [ 2.        , -2.        ,  1.        ],
       [-0.5       ,  1.        , -0.5       ]])
```

Ainsi, on a remplacé la composante $(2, 3)$ de `mat1` (soit 5) par 6, puis on a demandé l'inverse de la matrice modifiée. On peut obtenir la transposée de `mat1` en faisant : `np.transpose(mat1)`, et son déterminant en faisant : `np.linalg.det(mat1)`.

— On fait :

```
def polynome(x):
    return (x**2 - 2*x + 1)
```

```
polynome(3.5)
```

Cela renvoie : `6.25`. On a ainsi créé une fonction `polynome` qui correspond à $p(x) = x^2 - 2x + 1$, et demandé la valeur de celui-ci en $x = 3.5$. On peut créer un large panel de fonctions avec cette syntaxe. Attention, l'indentation est importante ; Il doit y avoir un décalage dans les lignes écrites dans le corps de la fonction, sinon elles ne seront pas prises en compte dans l'exécution.

— On fait :

```
i = 0
while i < 3 :
   print("OK j'ai compris")
   i = i + 1
```

Cela renvoie :

```
OK j'ai compris
OK j'ai compris
OK j'ai compris
```

Ainsi, on a créé une boucle "tant que", et plus précisément : "tant qu'une condition n'est pas vérifiée, faire cela". Attention, tout comme la définition d'une fonction, l'indentation est importante pour tous les types de boucles.

— On fait :

```
i = 0
for i in range(0, 3) :
   print("OK j'ai encore compris")
   i = i + 1
```

Cela renvoie :

```
OK j'ai encore compris
OK j'ai encore compris
OK j'ai encore compris
```

Ainsi, on a créé une boucle "pour", et plus précisément : "pour un indice allant de tant à tant, faire cela".

— On fait :

```
a = 2
if a > 5 :
    print("OK c'est clair")
else :
    print("OK c'est très clair")
```

Cela renvoie :

```
OK c'est très clair
```

Ainsi, on a créé une boucle “si”, et plus précisément : “si une condition est vérifiée, faire cela, sinon, faire autre chose”.

Pour allez plus loin, voir les fonctions `map`, `filter`, `lambda`, `apply`, etc.

En fait, il y aurait encore beaucoup à faire pour tester toutes les commandes intéressantes de Python, mais les opérations ci-dessus suffisent pour notre objectif d'analyse statistique, sachant que d'autres nouvelles commandes seront présentées au fil du document.

1.3 Tableau de données

Il est important de savoir manipuler des données, particulièrement celles mises sous la forme d'un tableau. Le module `pandas` (déjà chargé) est fait pour cela. On peut aussi faire avec `numpy`, mais `pandas` est plus adapté à l'analyse statistique poussée. Nous allons voir les bases, et procéder avec des exemples.

On commence par créer un tableau de données simple en faisant :

```
t = np.arange(0,10,0.1)
x = np.sin(t)
y = np.random.choice(["A", "B", "C"], 100)
dataset = pd.DataFrame({"Time" : t, "x" : x, "y" : y})
```

Ainsi,

- `t` définit un vecteur contenant les valeurs 0, 0.1, 0.2, . . . , 9.8, 9.9. Il y a donc 100 valeurs (chiffre que l'on peut retrouver en faisant `np.shape(t)`).
- `t` est un vecteur contenant les valeurs de $\sin(t)$ pour les valeurs de t, donc $\sin(0)$, $\sin(0.1)$, $\sin(0.2)$, . . . , $\sin(9.8)$, $\sin(9.9)$.
- `y` est un vecteur à 100 composantes, dont chacune des composantes contient une des modalités choisies au hasard parmi "A", "B" et "C".
- `dataset` est le tableau de données constitué des colonnes `t` appelée `Time`, `x` appelé `x` et `y` appelé `y`.

En toute circonstance, dans un premier temps, on peut demander une entête de `dataset` en faisant :

```
dataset.head()
```

Cela renvoie :

```
   Time         x  y
0   0.0  0.000000  C
1   0.1  0.099833  C
2   0.2  0.198669  A
3   0.3  0.295520  C
4   0.4  0.389418  C
```

Si l'on ne dispose pas des informations sur la nature des données, on peut faire :

```
dataset.info()
```

Cela renvoie :

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Time    100 non-null    float64
 1   x       100 non-null    float64
 2   y       100 non-null    object
dtypes: float64(2), object(1)
memory usage: 2.5+ KB
```

Ainsi, il y a 100 lignes et 3 colonnes, `float64` précise que les données associées sont numériques, et `object` précise que les données associées sont non-numériques (caractères, etc.).

La suite de cette section concerne la manipulation du jeu de données `dataset`.

1.4 Manipulations d'un tableau de données

Partant de `dataset`, on peut faire les manipulations suivantes.

— On peut extraire la colonne `y` en faisant :

```
dataset["y"]
```

Cela renvoie :

```
0     C
1     C
2     A
3     C
4     C
     ..
95    A
96    C
97    C
98    A
99    B
Name: y, Length: 100, dtype: object
```

On aurait aussi pu faire : `dataset.y`

— On peut extraire les colonnes `x` et `y` en faisant :

```
dataset[["x", "y"]]
```

Cela renvoie :

```
           x  y
0   0.000000  C
1   0.099833  C
2   0.198669  A
3   0.295520  C
4   0.389418  C
..       ... ..
95 -0.075151  A
96 -0.174327  C
97 -0.271761  C
98 -0.366479  A
99 -0.457536  B

[100 rows x 2 columns]
```

— On peut extraire des lignes en les appelant par leurs indices en faisant, par exemple :

```
dataset[5:10]
```

Cela renvoie :

```
   Time         x  y
5   0.5  0.479426  A
6   0.6  0.564642  C
7   0.7  0.644218  A
8   0.8  0.717356  A
9   0.9  0.783327  B
```

On ainsi extrait la sixième ligne en premier (Python commençant à l'indice 0), jusqu'à la dixième.

— On peut extraire en même temps des colonnes et des lignes en faisant, par exemple :

```
dataset[["x", "y"]][5:10]
```

Cela renvoie :

```
          x  y
5  0.479426  A
6  0.564642  C
7  0.644218  A
8  0.717356  A
9  0.783327  B
```

On aurait aussi pu faire `df.iloc[5:10,[1,2]]`.

— On peut créer une nouvelle colonne en faisant, par exemple :

```
dataset["1000x"] = 1000 * dataset["x"]
```

On la supprime en faisant :

```
del dataset["1000x"]
```

— On peut compter les modalités d'un caractère qualitatif en faisant, par exemple :

```
dataset["y"].value_counts()
```

Cela renvoie :

```
C    38
A    31
B    31
Name: y, dtype: int64
```

Ainsi, il y a 38 fois la modalité `C`, 31 fois la modalité `A` et 31 fois la modalité `B`.

— On peut créer un sous-jeu de données en fonction d'une ou plusieurs modalités d'un caractère qualitatif en faisant, par exemple :

```
dataset_A = dataset.loc[dataset.y == "A"]
```

Ainsi, on a sélectionné toutes les lignes de `dataset` dans lesquels `y` affiche `A`.

Pour créer un jeu de données constitué des lignes de `dataset` dans lesquelles `y` affiche `A` ou `B`, on fait :

```
dataset_AB = dataset.loc[(dataset.y == "A") | (dataset.y == "B")]
```

— Pour aller plus loin, il faudrait aborder, entre autre, le changement de noms des colonnes si besoin est : `dataset.rename(columns = {"Time" : "Temps"})`, la gestion des valeurs manquantes s'il y a : `dataset.dropna()`, la maîtrise de la fonction `apply`, etc.

1.5 Importation de données

On peut importer un tableau de données en fonction de son format avec les commandes `pd.read_csv` pour un fichier `text` ou `csv`, ou `pd.ExcelFile`, `xls.parse` ou `pd.read_excel`. Dès lors, il faut préciser l'endroit ou se trouve le tableau, ou bien, saisir l'url où se situe le tableau, préciser quels sont les séparateurs entre les données ou les colonnes (option `sep`), s'il y a le nom au colonnes (option `header`), s'il y a des valeurs manquantes ou non (option `na_values`).

Un exemple qui sera étudié dans le chapitre suivant est :

```
tips = pd.read_csv("https://chesneau.users.lmno.cnrs.fr/tips.csv",
header = 0, sep = ",")
```

2 Lois de probabilité

Cette section montre comment on peut manipuler les lois de probabilité avec Python.

2.1 Loi normale

On commence avec la principale loi de probabilité à densité : la loi normale. Les commandes pourront s'adapter sans efforts à d'autres loi à densité, qui seront listées à la fin de cette section. Toute la documentation sur la loi normale est disponible en faisant : `print(stats.norm.__doc__)`

2.1.1 Densité

La densité de la loi normale centrée réduite, notée $\mathcal{N}(0,1)$, est donnée par

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}, \quad x \in \mathbb{R}.$$

La commande associée est `stats.norm.pdf(x)`.

Le terme "pdf" en anglais désigne "probability density function", soit le plus court "densité" en français.

Par exemple, pour calculer sa valeur en $x = 0$, on fait :

```
stats.norm.pdf(0)
```

Cela renvoie : `0.3989422804014327`. Cela correspond à $1/\sqrt{2\pi}$.

Pour calculer les valeurs de cette densité en plusieurs points en seule fois, on peut faire :

```
stats.norm.pdf([-1.5, -0.05, 0.55, 1.8])
```

Cela renvoie : `array([0.1295176 , 0.39844391, 0.34294386, 0.07895016])`

Donc, par analogie, on a par exemple : $f(-0.05) = 0.39844391$.

On veut vérifier que l'intégrale de $f(x)$ pour $x \in \mathbb{R}$ fait bien 1, avec les commandes suivantes :

```
def normal_distribution_function(x) :
    value = scipy.stats.norm.pdf(x)
    return value


scipy.integrate.quad(normal_distribution_function, -np.inf, np.inf)
```

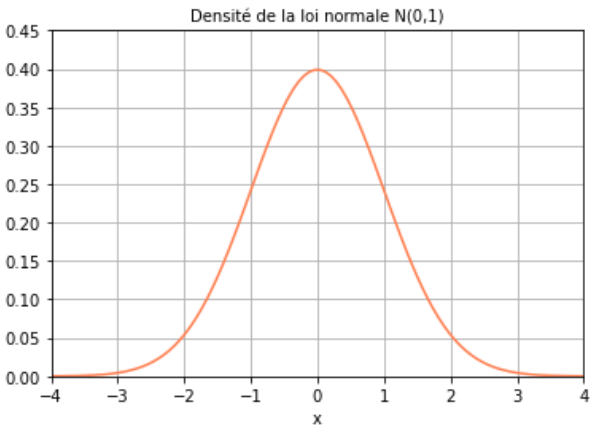
Cela renvoie : `(0.9999999999999998, 1.0178191349259989e-08)`

Le premier chiffre donnant une valeur approchée de l'intégrale, quasiment 1 ici, et le deuxième donnant l'erreur d'approximation, très faible ici.

On peut tracer le graphique de $f(x)$ en faisant :

```
x_min = -4
x_max = 4

x = np.linspace(x_min, x_max, 100)

y = scipy.stats.norm.pdf(x)

plt.plot(x,y, color = "coral")

plt.grid()

plt.xlim(x_min,x_max)
plt.ylim(0,0.45)

plt.title("Densité de la loi normale N(0,1)", fontsize = 10)

plt.xlabel("x")
```

Cela renvoie :



Plus généralement, la densité de la loi normale (pas forcément centrée, ni réduite), notée $\mathcal{N}(\mu, \sigma^2)$, est donnée par

$$f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, \quad x \in \mathbb{R}.$$

La commande associée est `stats.norm.pdf(x, mu, sigma)`.

Par exemple, pour calculer sa valeur en $x = 0$, avec $\mu = 1$ et $\sigma = 2$, on fait :

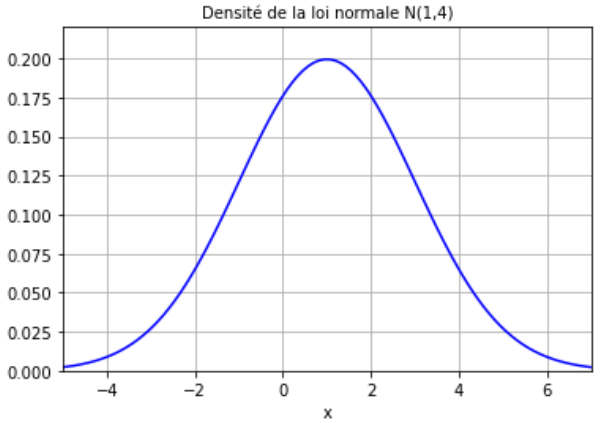
```
stats.norm.pdf(0, 1, 2)
```

Cela renvoie : `0.17603266338214976`

On peut tracer le graphique de $f(x;\mu,\sigma)$, avec $\mu = 1$ et $\sigma = 2$ par exemple, en faisant :

```
x_min = -5
x_max = 7

x = np.linspace(x_min, x_max, 100)

y = scipy.stats.norm.pdf(x, 1, 2)

plt.plot(x,y, color = "blue")

plt.grid()

plt.xlim(x_min,x_max)
plt.ylim(0,0.22)

plt.title("Densité de la loi normale N(1,4)", fontsize = 10)

plt.xlabel("x")
```

Cela renvoie :



2.1.2 Fonction de répartition

La fonction de répartition de la loi normale centrée réduite $\mathcal{N}(0,1)$ est donnée par

$$\Phi(x) = \int_{-\infty}^{x} f(t)dt = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{t} e^{-\frac{t^2}{2}} dt, \quad x \in \mathbb{R}.$$

Elle n'a pas de forme analytique simple. La commande associée est `stats.norm.cdf(x)`.

Le terme "cdf" en anglais désigne "cumulative distribution function", ce qui équivaut à la "fonction de répartition" en français.

Par exemple, pour calculer sa valeur en $x = 0$, on fait :

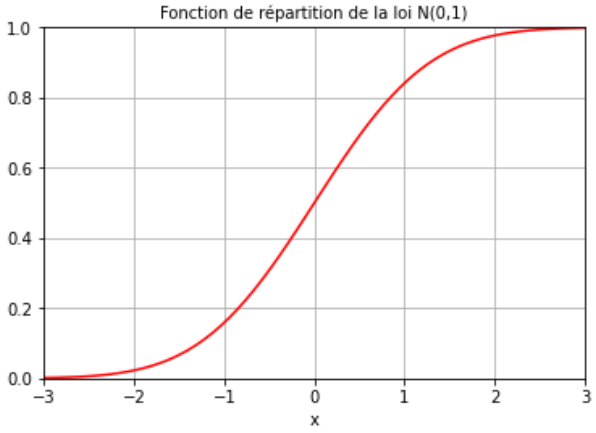
```
stats.norm.cdf(0)
```

Cela renvoie : `0.5`.

On peut tracer le graphique de cette fonction de répartition en faisant :

```
x_min = -3
x_max = 3

x = np.linspace(x_min, x_max, 100)

y = scipy.stats.norm.cdf(x)

plt.plot(x, y, color = "red")

plt.grid()

plt.xlim(x_min,x_max)
plt.ylim(0,1)

plt.title("Fonction de répartition de la loi N(0,1)", fontsize = 10)

plt.xlabel("x")
```

Cela renvoie :



Plus généralement, la fonction de répartition de la loi normale $\mathcal{N}(\mu, \sigma^2)$, est donnée par

$$\Phi(x; \mu, \sigma) = \int_{-\infty}^{x} f(t; \mu, \sigma)dt = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2} dt, \quad x \in \mathbb{R}.$$

La commande associée est `stats.norm.cdf(x, mu, sigma)`.

Par exemple, pour calculer sa valeur en $x = 0$, avec $\mu = 1$ et $\sigma = 2$, on fait :

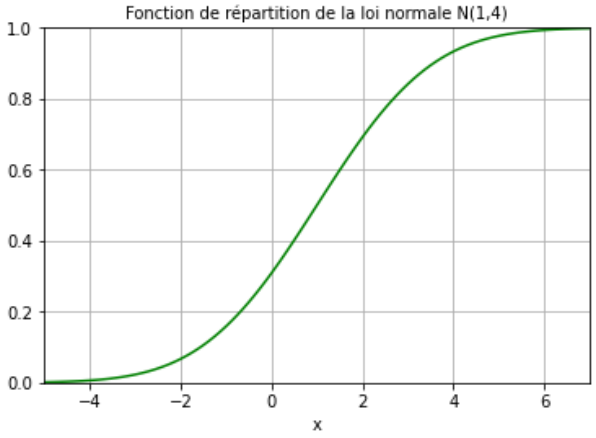
```
stats.norm.cdf(0, 1, 2)
```

Cela renvoie : `0.3085375387259869`

On peut tracer le graphique de $F(x; \mu, \sigma)$, avec $\mu = 1$ et $\sigma = 2$ par exemple, en en faisant :

```
x_min = -5
x_max = 7

x = np.linspace(x_min, x_max, 100)

y = scipy.stats.norm.cdf(x, 1, 2)

plt.plot(x, y, color = "green")

plt.grid()

plt.xlim(x_min,x_max)
plt.ylim(0,1)

plt.title("Fonction de répartition de la loi normale N(1,4)", fontsize = 10)

plt.xlabel("x")
```

Cela renvoie :



2.1.3 Fonction de quantile

La fonction de quantile est définie comme l'inverse de la fonction de répartition. La fonction de quantile de la loi normale $\mathcal{N}(\mu, \sigma^2)$ est donnée par

$$Q(x; \mu, \sigma) = \Phi^{-1}(x; \mu, \sigma), \quad x \in [0, 1]$$

Elle n'a pas d'expression analytique simple. La commande associée est `stats.norm.ppf(x, mu, sigma)`.

Le terme "ppf" en anglais désigne "percent point function", ce qui équivaut à la "fonction de quantile" en français.

Elle permet, entre autres, de calculer les quartiles de la loi normale. Pour la loi normale centrée réduite, on fait :

```
stats.norm.ppf([0.25, 0.5, 0.75])
```

Cela renvoie : `array([-0.67448975, 0. , 0.67448975])`. Ainsi, le premier quartile vaut -0.67448975, le deuxième correspondant à la médiane vaut 0, et le troisième vaut 0.67448975.

2.1.4 Generation de valeurs

Partant d'une variable aléatoire X suivant la loi normale $\mathcal{N}(\mu, \sigma^2)$, on est en mesure de générer des données pouvant être des observations de celle-ci.

La commande associée est : `stats.norm.rvs(size = nombredevaleurs, loc = mu, scale = sigma)`.

Le terme "rvs" en anglais désigne "random values", pour valeurs aléatoires.

Ainsi, par exemple, pour générer 100 valeurs d'une variable aléatoire X suivant la loi $\mathcal{N}(0, 1)$, on fait :

```
stats.norm.rvs(size = 100)
```

Cela renvoie :

```
array([ 0.25067356,  0.57109441, -2.15560285,  0.66154265, -0.50940351,
       -1.29257668, -0.16472959, -0.83399851,  0.63307235, -0.87908076,
       -1.88421464,  1.68993283, -0.10951058,  1.20713559,  1.13787113,
        1.34553723,  2.23830867, -1.01218016,  0.08428463, -0.26362582,
        0.94384107, -0.86918136, -1.18452049,  0.08495551, -0.39732404,
       -0.97807379, -0.02647738,  2.37889813,  0.44343894, -0.86882942,
        1.73723   ,  2.50049059,  1.66530554, -1.36996373,  0.02811513,
        0.47256107, -0.5119904 ,  0.21660877,  0.88491503,  0.83056259,
        0.7095139 ,  0.24633787, -0.47999708,  0.34213816,  2.08064348,
        1.03349184, -0.98082941, -0.0073745 ,  0.74535508,  0.23289402,
       -0.99917729, -0.24636614,  0.04439759, -0.68667832, -1.84142986,
       -2.31567139,  0.94739266, -0.82011736, -2.00424583,  1.11649194,
       -1.04400852,  2.08725705, -0.13993375,  0.33913452, -0.72056726,
        1.88300757,  1.25450227,  0.01853906,  0.37537415,  1.08031337,
       -0.09905388,  0.81953011, -0.57846215, -0.76569758, -0.4938571 ,
        0.33195583,  1.73353366, -1.64586939, -0.76101448, -0.91836999,
       -1.22224172,  1.18999571, -0.95344622, -1.15956456, -0.06275574,
       -2.20259188, -0.21379541, -0.20147497,  0.46174247,  0.63012475,
       -0.83380484, -0.64761551,  1.30675537, -0.82359776, -0.15987368,
       -1.39727748,  0.06027738, -1.44367325,  1.23420259, -0.52322195])
```

A chaque expérience, on aura d'autres valeurs dues au caractère aléatoire du processus de génération.

Un autre, pour générer 100 valeurs d'une variable aléatoire X suivant la loi $\mathcal{N}(1, 4)$, on fait :

```
stats.norm.rvs(size = 100, loc = 1, scale = 2)
```

Cela renvoie :

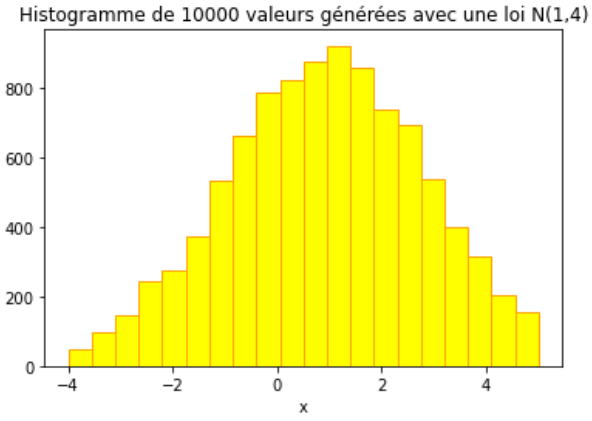
```
array([ 1.54155622, -2.38967674,  0.83657875,  2.56081313,  0.84571225,
        2.24857733,  1.61992423, -0.39916374,  3.57804047, -1.54829371,
        1.67942564,  0.2169668 , -2.11530843, -0.1023769 ,  1.03744378,
       -1.5631107 ,  0.46745435,  3.55444575, -0.19764676,  0.30323231,
        1.67817203, -0.82981046,  0.92431526,  0.59204893,  0.78396422,
        1.85353875, -1.12376593,  0.96345763,  2.51878492,  2.13452503,
        4.63783803, -0.41250713,  0.00705451,  3.82765217,  1.13103313,
        0.99358171, -2.23959789,  1.5080891 ,  0.56940911,  1.68118263,
        2.87098694,  1.28419714, -0.57135961,  2.74881605, -0.67965736,
        4.8289641 ,  1.84501237,  1.42732064, -0.62854287, -1.91407402,
        0.60836269, -1.00758467, -0.57676664,  2.22790311, -4.46212831,
```

```
 1.05133282,  1.08519288,  1.34864986, -0.63205573, -1.29892085,
 2.66127328,  1.77207929,  0.44200818, -0.79814358,  1.11891043,
 1.44594771,  5.52309882, -0.86107218, -0.57484911, -0.24056311,
-0.45170112, -0.41513345, -1.06186337,  4.53664799,  1.10702723,
 1.71535688,  3.48339343,  1.92367472,  0.65179746,  1.04511804,
-0.73358224,  2.1705509 , -0.12028903,  2.40380486,  2.06403127,
-1.10421038,  3.06956438,  2.84657427, -0.05558804,  0.85299668,
-1.20191256,  2.71297459,  0.829274  ,  1.28389021, -1.34370136,
 1.58431055, -0.71813228,  2.24073271,  2.56511162,  0.22762636])
```

Pour une visualisation de telles valeurs, on peut utiliser un histogramme. Par exemple, à partir de 10000 valeurs générées à partir d'une variable aléatoire X suivant la loi normale $\mathcal{N}(1,4)$, on peut faire :

```
x = stats.norm.rvs(size = 10000, loc = 1, scale = 2)


plt.hist(x, range = (-4, 5), bins = 20, color = "yellow", edgecolor = "orange")
plt.xlabel("x")
plt.title("Histogramme de 10000 valeurs générées avec une loi N(1,4)")
```

Cela renvoie :



Sans surprise, l'allure de l'histogramme est en forme de cloche.

2.1.5 Mesures de moments

Soit X un variable aléatoire suivant la loi normale $\mathcal{N}(\mu, \sigma^2)$. Alors on sait que son espérance, variance et écart-type sont

$$\mathbb{E}(X) = \mu, \quad \mathbb{V}(X) = \sigma^2, \quad \sigma(X) = \sigma,$$

respectivement. Partant de valeurs précise pour μ et σ, on peut retrouver ces valeurs avec les commandes : `stats.norm.mean(loc = mu, scale = sigma)`, `stats.norm.var(loc = mu, scale = sigma)` et `stats.norm.std(loc = mu, scale = sigma)`, respectivement. Le terme "mean" désigne "moyenne", le terme "var" désigne "variance" et le terme "std" désigne "standard deviation".

Dans le cadre d'une loi normale, ces commandes sont inutiles. Toutefois, elles se transposent à d'autres lois de probabilité dont l'espérance, variance et écart-type reposent sur des formules moins immédiates. Il faut donc les connaître à toute fin utile.

Ainsi, par exemple, on fait :

```
stats.norm.mean(loc = 1, scale = 2)
```

Cela renvoie sans surprise : `1.0`.

On peut vérifier cela à l'aide d'un intégration en faisant :

```
def x_normal_distribution_function(x) :
    value = x * scipy.stats.norm.pdf(x, loc = 1, scale = 2)
    return value

scipy.integrate.quad(x_normal_distribution_function, -np.inf, np.inf)
```

Cela renvoie : `(0.9999999999999999, 1.4514809962936249e-08)`

On retrouve la valeur 1, première composante du vecteur.

De plus, par exemple, on fait :

```
stats.norm.var(loc = 1, scale = 2)
```

Cela renvoie sans surprise : `4.0`.

On peut vérifier cela à l'aide d'un intégration en faisant :

```
def v_normal_distribution_function(x) :
    value = (x - 1)**2 * scipy.stats.norm.pdf(x, loc = 1, scale = 2)
    return value

scipy.integrate.quad(v_normal_distribution_function, -np.inf, np.inf)
```

Cela renvoie : `(4.000000000000003, 2.3960310858021453e-08)`

On retrouve la valeur 4, première composante du vecteur.

Précisons que l'on peut aussi avoir en une fois la moyenne, la variance, le coefficient de symétrie (skewness en anglais) et le coefficient d'aplatissement (kurtosis en anglais) avec les commandes :

```
stats.norm.stats(loc = mu, scale = sigma, moments = "mvsk")
```

Le "mvsk" étant pour "mean", "variance", "skewness" et "kurtosis".

2.1.6 Complements

— On peut définir une loi de probabilité en amont pour simplifier les commandes à venir en faisant, par exemple :
`dist = stats.norm(loc = 1, scale = 3)`
Dès lors, on définit la loi normale $\mathcal{N}(1, 9)$. On peut manipuler ces caractéristiques avec des commandes du type : `dist.pdf(0)`, `dist.cdf([0, 3, 6])`, `dist.rvs(size = 1000)`, etc.
— La plupart des lois à densité usuelles sont disponibles dans Python. Les principales sont :
— `stats.uniform(a, b)` : Loi uniforme dans l'intervalle $[a, b]$,
— `stats.t(m)` : loi de Student à m degrés de liberté,
— `stats.beta(a, b)` : loi beta de paramètres a et b,
— `stats.chi2(m)` : loi du chi2 à m degrés de liberté,
— `stats.expon(scale = 1 / v)` : loi exponentielle de paramètre v, i.e., de densité $f(x; v) = ve^{-vx}$, $x \geq 0$.
— `stats.gamma(a, scale = 1 / v)` : loi gamma de paramètres a et v, i.e., de densité $f(x; a, v) = (v^a/\Gamma(a))x^{a-1}e^{-vx}$, $x \geq 0$.

2.2 Loi de Poisson

Les lois discrètes sont également disponibles dans Python. Pour commencer, on ne présente ici que la loi de Poisson. Les commandes pourront s'adapter sans efforts à d'autres lois à densité, qui seront listées à la fin de cette section. Toute la documentation sur la loi normale est disponible en faisant : `print(stats.poisson.__doc__)`

2.2.1 Probabilité de masse

La probabilité de masse de la loi de Poisson de paramètre λ, notée $\mathcal{P}(\lambda)$, est donnée par

$$p(x; \lambda) = e^{-\lambda}\frac{\lambda^x}{x!}, \quad x \in \mathbb{N}.$$

La commande associée est `stats.poisson.pmf(x, lambda)`.

Le terme "pmf" en anglais désigne "probability mass function", soit "probabilité de masse" en français.

Par exemple, pour calculer sa valeur en $x = 0$ avec $\lambda = 2$ par exemple, on fait :
`stats.poisson.pmf(0, 2)`

Cela renvoie : `0.1353352832366127`. Cela correspond à $p(0; 2)$.

Pour calculer les valeurs de cette probabilité de masse en plusieurs points en seule fois, on peut faire :

```
stats.poisson.pmf([0, 3, 5, 9], 2)
```

Cela renvoie : `array([0.13533528, 0.18044704, 0.03608941, 0.00019095])`

Donc, par analogie, on a par exemple : $p(5;2) = 0.03608941$.

On veut vérifier que la somme des valeurs de $p(x;\lambda)$ pour $x \in \mathbb{N}$, avec $\lambda = 2$ par exemple, fait bien 1, avec les commandes suivantes :

```
a = scipy.stats.poisson.pmf(list(range(100)), 2)


np.sum(a)
```

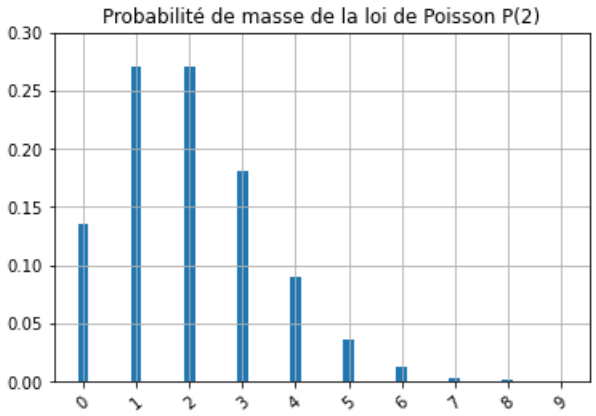
Cela renvoie : `1.0`

On a choisi la limite de 99 car, au-delà, les valeurs de la probabilité de masse est extrêmement petite ; une série infinie étant difficilement prise en compte par Python.

On peut tracer le graphique en bâton de $p(x;\lambda)$, avec $\lambda = 2$ par exemple, en faisant :

```
x = list(range(10))
height = scipy.stats.poisson.pmf(x, 2)
width = 0.2

plt.bar(x, height, width)

plt.ylim(0,0.3)
plt.grid()

plt.title("Probabilité de masse de la loi de Poisson P(2)")

pylab.xticks(x,  rotation = 40)
```

Cela renvoie :



2.2.2 Fonction de répartition

La fonction de répartition de la loi de Poisson $\mathcal{P}(2)$ est donnée par

$$F(x;\lambda) = \sum_{t=0}^{[x]} p(t;\lambda), \quad x \in \mathbb{R},$$

où $[x]$ désigne la partie entière de x.

Elle n'a pas de forme analytique simple. La commande associée est `stats.poisson.cdf(x, lambda)`.

Par exemple, pour calculer sa valeur en $x = 3$, avec $\lambda = 2$ par exemple, on fait :

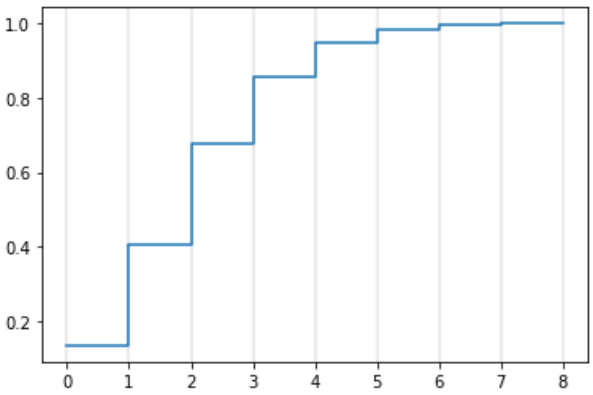
```
stats.poisson.cdf(3, 2)
```

Cela renvoie : `0.857123460498547`.

On peut tracer le graphique en escalier de cette fonction de répartition en faisant :

```
x = list(range(9))
cdf = scipy.stats.poisson.cdf(x,2)

for xc in x:plt.axvline(x = xc, color = "0.9")
plt.plot(x, cdf, drawstyle = "steps-post")
plt.xticks(x)
```

Cela renvoie :



2.2.3 Fonction de quantile

Dans le cadre d'une loi de Poisson, la fonction de quantile est définie par

$$Q(x;\mu,\sigma) = \inf\{k \in \mathbb{N};\ F(k;\lambda) \geq x\}.$$

Cette définition peut s'adapter à toute loi discrète. Elle n'a pas d'expression analytique simple. La commande associée est `stats.poisson.ppf(x, lambda)`.

Elle permet, entre autres, de calculer les quartiles de la loi de Poisson. Pour $\lambda = 2$, on fait :

```
stats.poisson.ppf([0.25, 0.5, 0.75], 2)
```

Cela renvoie : `array([1., 2., 3.])`. Ainsi, le premier quartile vaut 1, le deuxième correspondant à la médiane vaut 2, et le troisième vaut 3.

2.2.4 Generation de valeurs

Partant d'une variable aléatoire X suivant la loi de Poisson $\mathcal{P}(\lambda)$, on est en mesure de générer des données pouvant être des observations de celle-ci.

La commande associée est : `stats.poisson.rvs(lambda, size = nombredevaleurs)`.

Ainsi, par exemple, pour générer 100 valeurs d'une variable aléatoire X suivant la loi $\mathcal{P}(2)$, on fait :

```
stats.poisson.rvs(2, size = 100)
```

Cela renvoie :

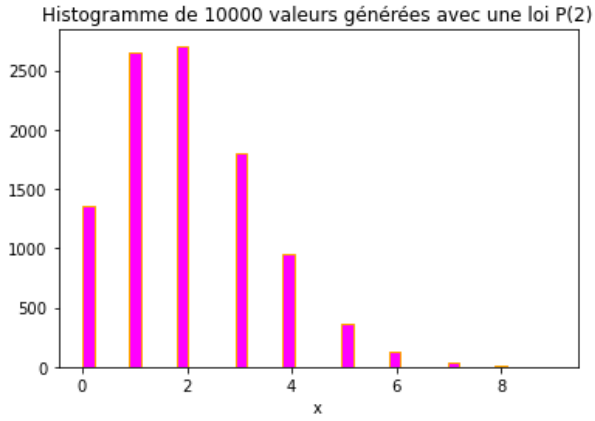
```
array([2, 1, 0, 4, 1, 3, 3, 2, 4, 2, 3, 1, 3, 1, 0, 3, 4, 3, 5, 1, 4, 1,
       1, 1, 3, 4, 5, 2, 3, 0, 2, 0, 3, 1, 5, 1, 2, 1, 1, 1, 1, 1, 4, 1,
       3, 3, 3, 2, 1, 1, 0, 1, 1, 1, 0, 3, 1, 5, 1, 2, 3, 2, 4, 0, 3, 1,
       3, 1, 1, 1, 2, 5, 4, 1, 3, 0, 1, 0, 2, 1, 0, 1, 1, 2, 1, 1, 1, 3,
       1, 3, 3, 4, 2, 3, 2, 1, 1, 2, 2, 1])
```

A chaque expérience, on aura d'autres valeurs dues au caractère aléatoire du processus de génération.

Pour une visualisation de telles valeurs, on peut utiliser un histogramme. Par exemple, à partir de 10000 valeurs générées d'une variable aléatoire X suivant la loi normale $\mathcal{N}(1,4)$, on peut faire :

```
x = stats.poisson.rvs(2, size = 10000)


plt.hist(x, range = (0, 9), bins = 40, color = "magenta", edgecolor = "orange")
plt.xlabel("x")
plt.title("Histogramme de 10000 valeurs générées avec une loi P(2)")
```

Cela renvoie :



2.2.5 Mesures de moments

Soit X un variable aléatoire suivant la loi de Poisson $\mathcal{P}(2)$. Alors on sait que son espérance, variance et écart-type sont

$$\mathbb{E}(X) = \lambda, \quad \mathbb{V}(X) = \lambda, \quad \sigma(X) = \sqrt{\lambda},$$

respectivement. Partant de valeurs précises pour λ, on peut retrouver ces valeurs avec les commandes : `stats.poisson.mean(lambda)`, `stats.poisson.var(lambda)` et `stats.poisson.std(lambda)`, respectivement.

Dans le cadre d'une loi de Poisson, tout comme la loi normale, ces commandes sont inutiles. Toutefois, elles se transposent à d'autres lois de probabilité dont l'espérance, variance et écart-type reposent sur des formules moins immédiates. Il faut donc les connaître à toute fin utile.

Ainsi, par exemple, on fait :

```
stats.poisson.mean(3)
```

Cela renvoie sans surprise : `3.0`.

On peut vérifier cela à l'aide d'une sommation en faisant :

```
x = list(range(100))

a = x * scipy.stats.poisson.pmf(x, 3)

np.sum(a)
```

Cela renvoie : `3.000000000000001`

De plus, par exemple, on fait :

```
stats.poisson.var(3)
```

Cela renvoie sans surprise : `3.0`.

Précisons que l'on peut aussi avoir en une fois la moyenne, la variance, le coefficient de symétrie (skewness en anglais) et le coefficient d'aplatissement (kurtosis en anglais) avec les commandes :

```
stats.poisson.stats(loc = mu, scale = sigma, moments = "mvsk")
```

2.2.6 Complements

— On peut définir une loi de probabilité en amont pour simplifier les commandes à venir en faisant, par exemple :

```
dist = stats.poisson(3)
```

Dès lors, on définit la loi de Poisson $\mathcal{P}(3)$. On peut manipuler ces caractéristiques avec des commandes du type : `dist.pmf(0)`, `dist.cdf([0, 3, 6])`, `dist.rvs(size = 1000)`, etc.

— La plupart des lois discrètes usuelles sont disponibles dans Python. Les principales sont :

- `stats.bernoulli(p)` : loi de Bernoulli de paramètre p,
- `stats.binom(n, p)` : loi binomiale de paramètres n et p,
- `stats.nbinom(n, p)` : loi binomiale négative de paramètres n et p,
- `stats.geom(p)` : loi géométrique de paramètre p,
- `stats.hypergeom(M, n, N)` : loi hypergéometrique de paramètres n, p et N.

3 Données de référence : tips

Il s'agit désormais de voir des commandes Python permettant d'analyser des données de toutes sortes. Pour ce faire, nous allons procéder avec un exemple concret : dans le reste de ce document (hors exercices), on considère le jeu de données intitulé `tips`. Il est disponible ici :
`https://chesneau.users.lmno.cnrs.fr/tips.csv`

Dans un premier temps, on présente ce jeu de données. Un serveur a enregistré des informations sur chaque pourboire qu'il a reçu sur une période de quelques mois de travail dans un restaurant. Il a collecté plusieurs variables qui sont :

- `total_bill` (facture) en dollars : caractère quantitatif,
- `tip` (pourboire) en dollars : caractère quantitatif,
- `sex` (sexe) de celui qui a payé la facture : caractère qualitatif de modalités : Male et Female
- `smoker` (fumeur) indique s'il y avait des fumeurs dans la soirée : caractère qualitatif de modalités : Yes and No
- `day` (jour) de la semaine : caractère qualitatif de modalités : Sun, Sat, Fri, et Thur,
- `time` (moment) de la journée : caractère qualitatif de modalités : Lunch et Dinner,
- `size` (taille) de la table : caractère quantitatif discret.

En tout, le serveur a enregistré 244 pourboires. Les données ont été rapportées dans une collection d'études de cas pour les statistiques d'entreprises. La référence de ce jeu de données est :

Bryant, P. G. and Smith, M (1995) Practical Data Analysis : Case Studies in Business Statistics. Homewood, IL : Richard D. Irwin Publishing.

L'intérêt majeur du jeu de données `tips` est qu'il contient toutes les natures des caractères possibles, avec des liens existants entre eux (sans valeurs manquantes, ou autres "problèmes"). Il est donc parfait pour une première analyse.

On rappelle que l'on suppose charger tous les modules nécessaires, à savoir :

```
import matplotlib.pyplot as plt
import scipy.stats
import numpy as np
import pylab
import pandas as pd
import seaborn as sns
import statistics
import statsmodels.api as smi
```

On charge les données en faisant :

```
tips = pd.read_csv("https://chesneau.users.lmno.cnrs.fr/tips.csv",
header = 0, sep = ",")
```

Une entête des premières lignes est disponible en faisant :

```
tips.head()
```

Cela renvoie :

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

On retrouve les noms de nos caractères, avec un aperçu rapide des données saisies.

Également, on peut avoir une description rapide du jeu de données en faisant :

```
tips.info()
```

Cela renvoie :

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    object
 3   smoker      244 non-null    object
 4   day         244 non-null    object
 5   time        244 non-null    object
 6   size        244 non-null    int64
dtypes: float64(2), int64(1), object(4)
memory usage: 13.5+ KB
```

On retrouve les informations dont on disposait déjà (nombre de données par caractère, nature des caractères, etc.)

Le reste du document concerne l'analyse de ces données. Pour ce faire, il est supposé que les notions suivantes sont connues un minimum : outils de statistiques descriptives, graphiques statistiques, tests statistiques et intervalles de confiance.

4 Statistique descriptive

Cette partie s'intéresse à la statistique descriptive des données de `tips` en prenant en compte la nature des caractères.

4.1 Description de caractères quantitatifs

On considère les caractères quantitatifs du jeu de données `tips`, à savoir : `total_bill`, `tip` et `size`. On a une analyse globale (ou résumé) des mesures statistiques de ces caractères en faisant :
`tips.describe()`

Cela renvoie :

```
       total_bill         tip        size
count  244.000000  244.000000  244.000000
mean    19.785943    2.998279    2.569672
std      8.902412    1.383638    0.951100
min      3.070000    1.000000    1.000000
25%     13.347500    2.000000    2.000000
50%     17.795000    2.900000    2.000000
75%     24.127500    3.562500    3.000000
max     50.810000   10.000000    6.000000
```

On a donc les moyennes, écart-types, minimums, premier quartiles, deuxième quartiles (médianes), troisième quartiles, et maximums des trois caractères.

On aurait pu faire une analyse locale des variables. Par exemple, avec la variable `total_bill`, on aurait pu faire :

```
tips.total_bill.describe()
```

Cela renvoie :

```
count    244.000000
mean      19.785943
std        8.902412
min        3.070000
25%       13.347500
50%       17.795000
75%       24.127500
max       50.810000
Name: total_bill, dtype: float64
```

Ou bien : `tips["total_bill"].describe()`.

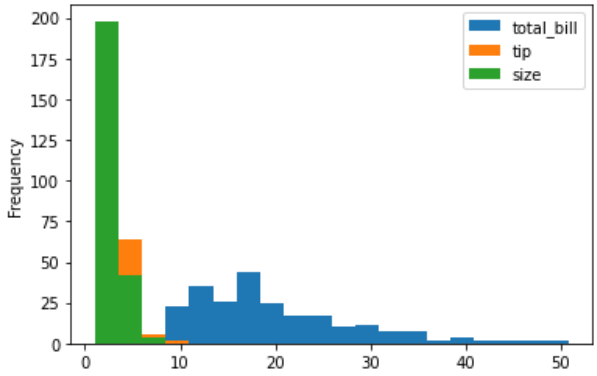
Les composantes de cette sortie s'obtiennent en faisant :
`tips.total_bill.mean()`, `tips.total_bill.std()`, `tips.total_bill.min()`,
`tips.total_bill.max()`, `tips.total_bill.median()` et
`tips.total_bill.quantile([0.25, 0.5, 0.75])`.

A cette liste, on peut rajouter `tips.total_bill.var()`.

En termes graphiques, les caractères quantitatifs peuvent se représenter avec un histogramme ou une boîte à moustaches. Pour les histogrammes, on peut faire une analyse globale en faisant :

```
tips.plot.hist(bins = 20)
```

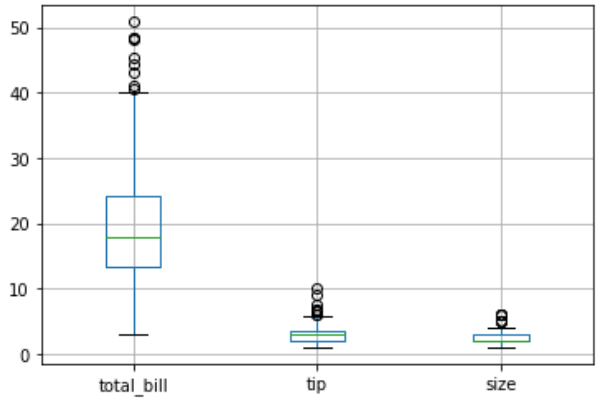
Cela renvoie :



Pour les histogrammes, on peut faire une analyse globale en faisant :

```
tips.boxplot()
```

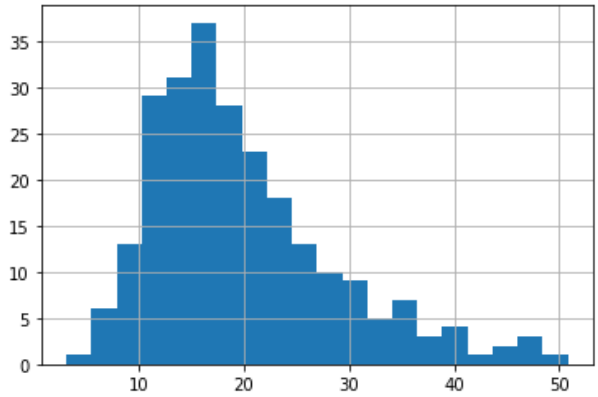
Cela renvoie :



On peut bien sûr faire une analyse graphique locale. Si on considère le caractère `total_bill`, on peut afficher son histogramme en faisant :

```
tips.total_bill.hist(bins = 20)
```

Cela renvoie :



On peut alors dire que la forme de l'histogramme est non-symétrique, ce qui présage une non-normalité des données. On approfondira ce point par la suite, avec le test de Shapiro-Wilk.

Pour avoir sa version normalisée, on fait :

```
tips.total_bill.hist(bins = 20, density = True)
```

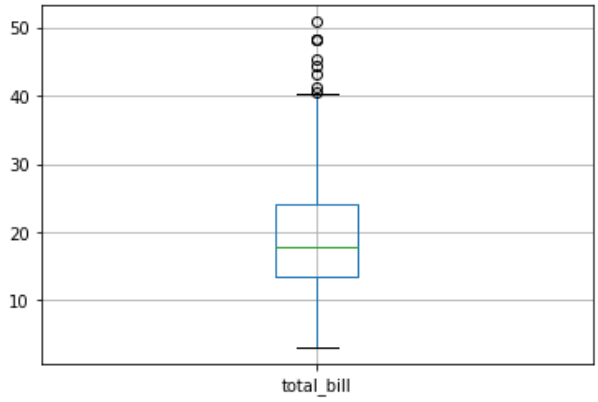
Ou bien, pour avoir le même en plus joli et avec la densité estimée d'affichée, on fait :

```
sns.distplot(tips.total_bill)
```

Si l'on veut la boîte à moustaches, on fait :

```
tips.boxplot(column = "total_bill")
```

Cela renvoie :



Ou bien, pour avoir le même en plus joli, on fait :

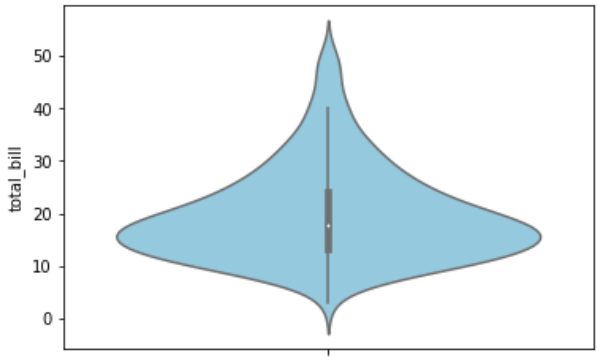
```
sns.boxplot(y = "total_bill", data = tips)
```

On constate plusieurs points dépassant la moustache supérieure, traduisant la présence de points que l'on peut considérer comme "anormaux" (ou extrêmes) par rapport aux autres. Cela va encore dans le sens d'une non-normalité des données.

Éventuellement, on peut afficher un graphique qui mélange histogramme et boîte à moustaches ; le graphique en violon. Pour ce faire, on fait :

```
sns.violinplot(y = "total_bill", data = tips, color = "skyblue")
```

Cela renvoie :



On y trouve une boîte à moustaches, et une figure symétrique par rapport à l'axe des ordonnées, donc chaque côté indique une estimation de la densité de l'histogramme (dont de la forme de l'histogramme).

L'analyse précédente peut se faire identiquement avec les caractères `tip` et `size`.

4.2 Description de caractères qualitatifs

On considère maintenant les caractères qualitatifs du jeu de données `tips`, à savoir : `sex`, `smoker`, `day` et `time`. On se focalise sur `sex` pour simplifier.

Dans un premier temps, on peut demander les modalités de `sex` en guise de rappel :

```
tips.sex.unique()
```

Cela renvoie :

```
array(["Female", "Male"], dtype = object)
```

On retrouve les modalités Female et Male, pour femme et homme.

On peut demander le tableau des effectifs en faisant :

```
pd.crosstab(tips.sex, "freq")
```

Cela renvoie :

```
col_0   freq
sex
Female    87
Male     157
```

Ainsi, dans l'échantillon de clients, il y a 87 femmes et 157 hommes.

On peut avoir le tableau des fréquences en faisant :

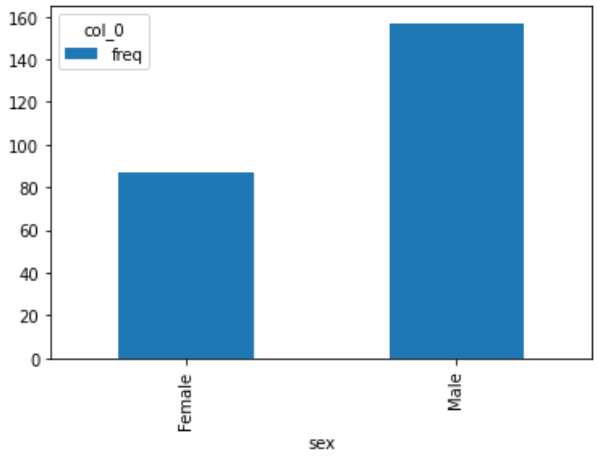
```
pd.crosstab(tips.sex, "freq", normalize = True)
```

On peut représenter les données de `sex` avec un diagramme en barres (ou bâtons) ou un diagramme circulaire.

Pour le diagramme en barres, on fait :

```
t = pd.crosstab(tips.sex, "freq")
t.plot.bar()
```

Cela renvoie :



En plus joli, on peut faire :

```
sns.countplot(x = "sex", data = tips)
```

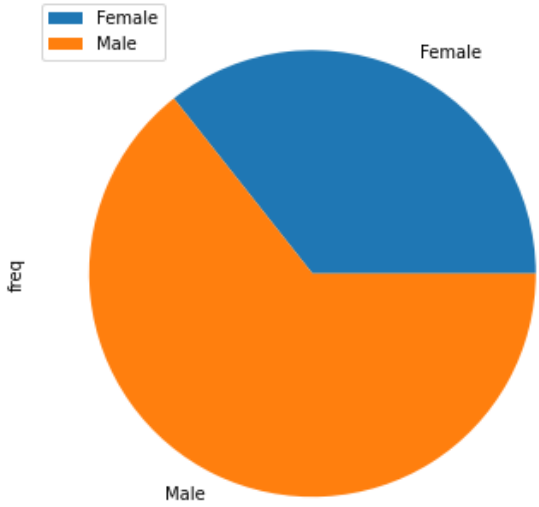
On peut faire de même avec les fréquences au lieu des effectifs, en faisant :

```
t = pd.crosstab(tips.sex, "freq", normalize = True)
t.plot.bar()
```

Pour obtenir un diagramme circulaire, on fait :

```
t = pd.crosstab(tips.sex, "freq")
t.plot.pie(subplots=True, figsize = (3, 3))
```

Cela renvoie :



Dans tous les cas, la surreprésentation des hommes est claire.

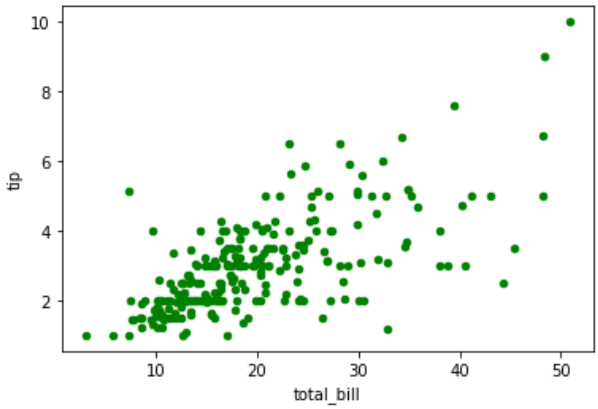
4.3 Description conjointe de caractères quantitatifs

On peut analyser conjointement les données de deux caractères quantitatifs, et, en particulier, un lien existant entre eux.

Pour ce faire, une représentation graphique à l'aide d'un nuage de points est utile. En considérant les caractères quantitatifs `total_bill` et `tip`, on fait :

```
tips.plot.scatter("total_bill", "tip", color = "green")
```

Cela renvoie :



On aurait aussi pu faire `plt.scatter(tips.total_bill, tips.tip, color = "green")`. Vu la forme allongée du nuage de points, une liaison linéaire est envisageable, même si celle-ci est visiblement modérée à cause d'une dispersion des points à mesure que les valeurs de `total_bill` croissent.

On peut mesurer le lien linéaire avec le coefficient de corrélation des données. On fait :

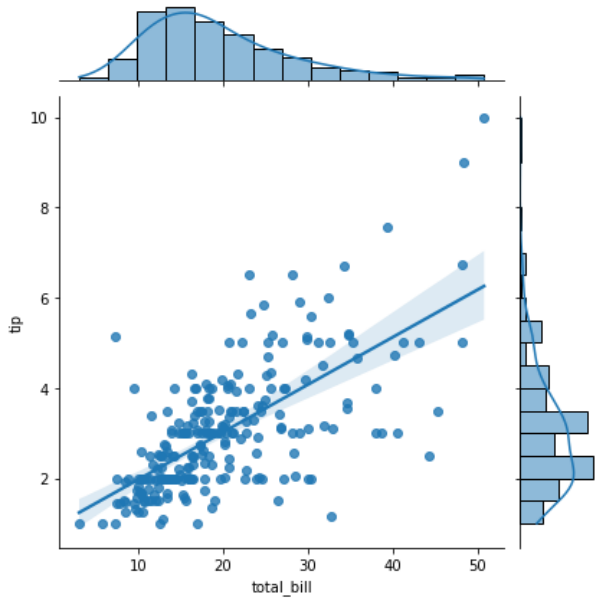
```
tips.total_bill.corr(tips.tip)
```

Cela renvoie : `0.6757341092113641`. Comme celui-ci appartient à l'intervalle $[0.5, 0.75]$, on peut effectivement parler de lien linéaire modéré. Cela sera affiné par la suite avec le test de corrélation de Pearson.

Pour une version graphique évoluée de cette analyse, on fait :

```
sns.jointplot(x = "total_bill", y = "tip", data = tips, kind = "reg")
```

Cela renvoie :



On dispose alors des histogrammes des données de chaque caractère, ainsi qu'une estimation de leur densité (forme), le nuage de points, et une droite l'ajustant "du mieux possible". La zone ombragée correspond à une zone de confiance dans laquelle la droite peut évoluer tout en gardant un ajustement correct (c'est une "zone de confiance").

4.4 Description conjointe de caractères qualitatifs

On peut analyser conjointement les données de deux caractères qualitatifs. Pour ce faire, on peut demander le tableau de contingence. En considérant les caractères qualitatifs `sex` et `smoker`, on fait :
`pd.crosstab(tips.sex, tips.smoker)`

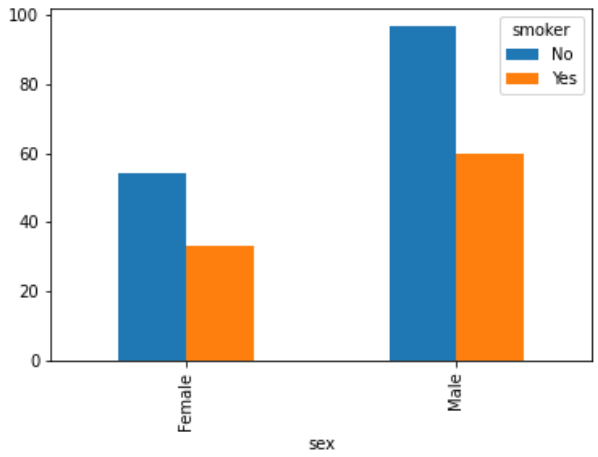
```
smoker  No  Yes
sex
Female  54   33
Male    97   60
```

Partant de ces effectifs, on pourra faire un test exact de Fisher ou du Chi-deux pour étudier un lien possible entre ces caractères.

Cela peut se représenter avec un diagramme en barres. On fait :

```
t = pd.crosstab(tips.sex, tips.smoker)
t.plot.bar()
```

Cela renvoie :



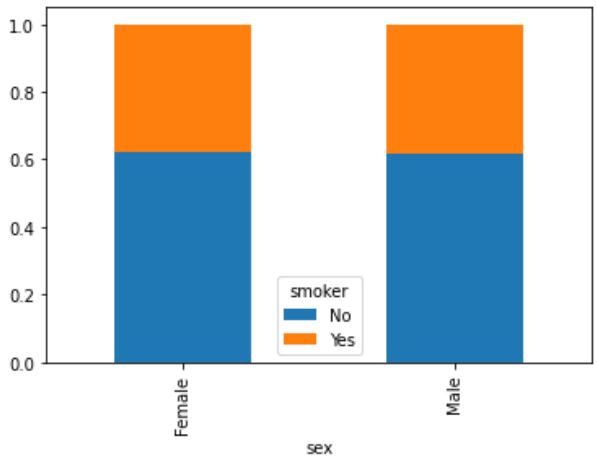
Pour avoir le diagramme des fréquences, on fait :

```
t = pd.crosstab(tips.sex, tips.smoker, normalize = True)
t.plot.bar()
```

Une version superposée est possible. On fait :

```
t = pd.crosstab(tips.sex, tips.smoker, normalize = "index")
t.plot.bar(stacked = True)
```

Cela renvoie :



On peut également comparer les diagrammes circulaires mis côte à côte en faisant :

```
t = pd.crosstab(tips.sex, tips.smoker)
t.plot.pie(subplots = True, figsize = (12, 6))
```

4.5 Description conjointe d'un caractère quantitatif et d'un caractère qualitatif

On peut analyser conjointement les données d'un caractère quantitatif et d'un caractère qualitatif. En particulier, les moyennes des caractères quantitatifs `total_bill`, `tip` et `size` en fonction des modalités de `sex` s'obtiennent en faisant :

```
tips.groupby("sex").mean()
```

Cela renvoie :

```
        total_bill       tip      size
sex
Female   18.056897  2.833448  2.459770
Male     20.744076  3.089618  2.630573
```

On constate que, à caractère quantitatifs fixé, les moyennes différent plus ou moins sensiblement en fonctions des modalités de `sex`.

Si on se focalise sur `total_bill`, on peut avoir une analyse descriptive plus détaillée en faisant :

```
tips.groupby("sex")["total_bill"].agg([np.mean, np.std, np.median, np.min, np.max])
```

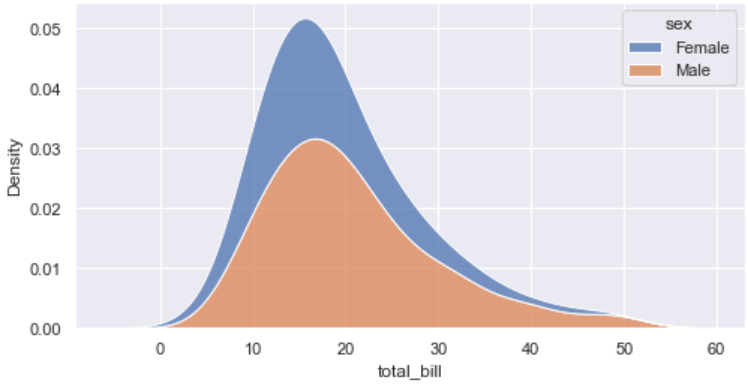
Cela renvoie :

```
            mean       std  median  amin   amax
sex
Female  18.056897  8.009209   16.40  3.07  44.30
Male    20.744076  9.246469   18.35  7.25  50.81
```

En termes graphique, on peut comparer l'allure des deux histogrammes associées aux deux modalités de `sex` en faisant :

```
sns.kdeplot(data = tips, x="total_bill", hue = "sex", multiple = "stack")
```

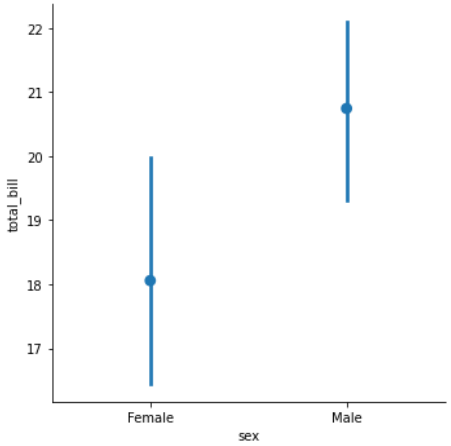
Cela renvoie :



Également, on peut visualiser la différence des moyennes avec le graphique des moyennes. On fait :

```
sns.catplot(x = "sex", y = "total_bill", data = tips, kind = "point", join = False)
```

Cela renvoie :

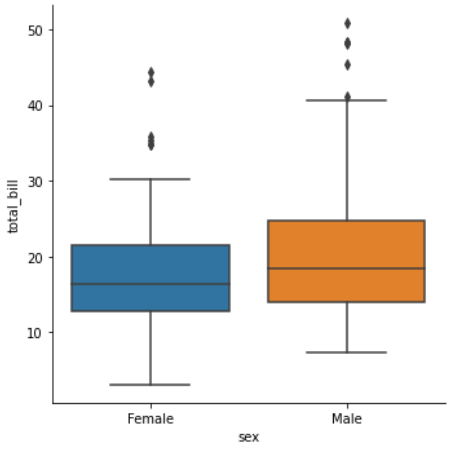


Les points représentent les moyennes de `total_bill` pour les modalités de `sex`. Les traits représentent des intervalles de confiance de ces moyennes.

On peut également comparer graphiquement les médianes en faisant :

```
sns.factorplot(x = "sex", y = "total_bill", data = tips, kind = "box")
```

Cela renvoie :



Pour faire avec des graphiques en violon, on fait :

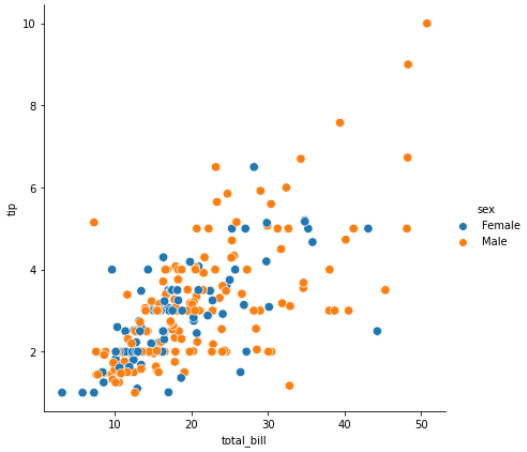
```
sns.factorplot(x = "sex", y = "total_bill", data = tips, kind = "violin")
```

4.6 Description conjointe de deux caractères quantitatifs et d'un caractère qualitatif

Un éventuel lien entre deux caractères quantitatifs en fonction des modalités de `sex` peut être analysé graphiquement. Pour ce faire, une représentation graphique à l'aide d'un nuage de points est utile. En considérant les caractères quantitatifs `total_bill` et `tip`, et le caractère quantitatif `sex`, on fait :

```
sns.relplot(x = "total_bill", y = "tip", hue = "sex", data = tips, height = 6, s = 70)
```

Cela renvoie :



On peut avoir plus complet en faisant :

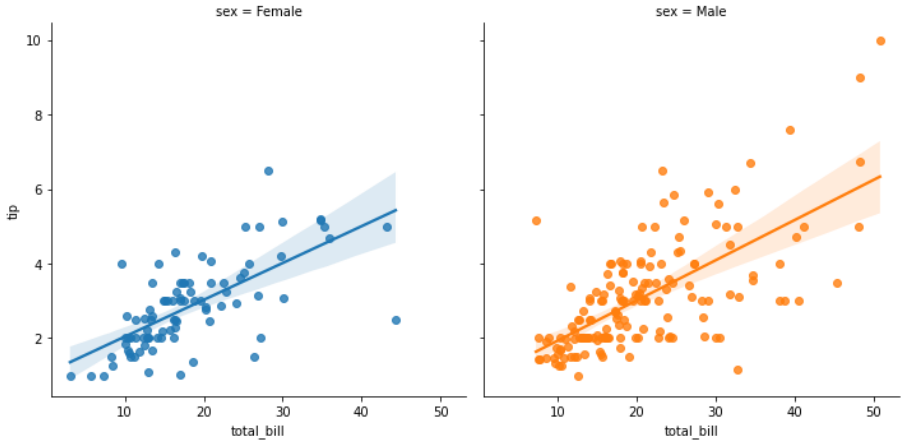
```
sns.jointplot(data = tips, x = "total_bill", y = "tip", marker = "*", hue = "sex")
```

On a donc deux sous-nuages de points correspondants aux deux modalités de `sex`, dans le nuage de points principal formé de `total_bill` et `tip`. Vu la forme allongée de ces deux nuages, une liaison linéaire est envisageable, suivant les modalités de `sex`.

Pour une meilleure visibilité, on fait :

```
sns.lmplot("total_bill", "tip", hue = "sex", col = "sex", data = tips)
```

Cela renvoie :



On a donc les nuages de points de `total_bill` et `tip` suivant les modalités de `sex`.

5 Tests statistiques

5.1 Test de la normalité

Partant des données, on souhaite tester la normalité de la distribution sous-jacente d'un caractère quantitatif. On parle aussi de normalité des données. Pour ce faire, on utilise le test de Shapiro-Wilk. La commande clé est `scipy.stats.shapiro`.

Pour travailler sur un exemple, on considère le caractère `total_bill`. Les hypothèses du test de la normalité sont de la forme : H_0 : "la distribution sous-jacente du caractère suit une loi normale" contre H_1 : "la distribution sous-jacente du caractère ne suit pas une loi normale". Pour admettre la normalité, on souhaite ne pas rejetter H_0 (donc si la p-valeur associée vérifie p-valeur > 0.05). Si H_0 est rejetée, l'hypothèse de normalité est rejetée également. Pour mettre en œuvre le test de Shapiro-Wilk, on fait :

```
scipy.stats.shapiro(tips.total_bill)
```

Cela renvoie :

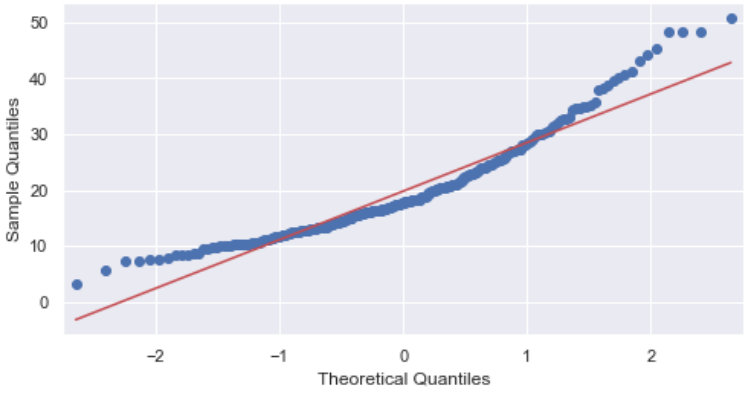
```
ShapiroResult(statistic=0.9197188019752502, pvalue=3.3245434183371003e-10)
```

Ainsi, on a p-valeur $= 3.3245434183371003 \times 10^{-10}$, donc p-valeur < 0.001 ; le rejet de la normalité des données de `total_bill` est hautement significatif $\star\star\star$. On avait déjà remarqué cela avec la forme de l'histogramme associée.

On peut visualiser cela avec le Q-Q plot en faisant :

```
smi.qqplot(tips.total_bill, line = "r")
```

Cela renvoie :



Clairement, le nuage de points n'est pas bien ajustable par une droite, traduisant la non-normalité des données.

En revanche, on peut voir si la distribution sous-jacente au logarithme de `total_bill`, caractère noté `log_total_bill` ; suit bien une loi normale. On fait :

```
log_total_bill = np.log(tips.total_bill)
scipy.stats.shapiro(log_total_bill)
```

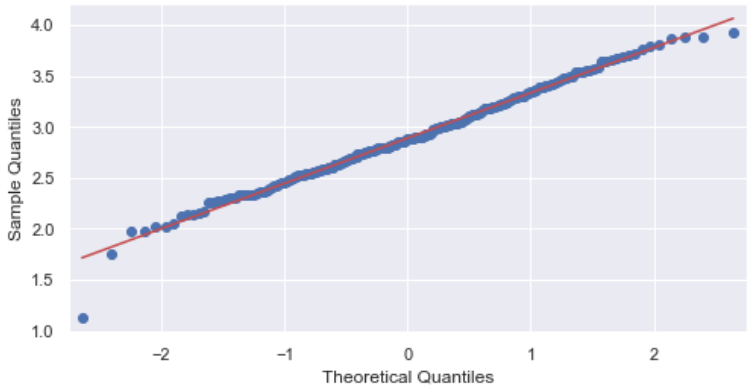
Cela renvoie :

```
ShapiroResult(statistic=0.9913218021392822, pvalue=0.15760214626789093)
```

Ainsi, on a p-valeur = 0.15760214626789093, donc p-valeur > 0.05 ; la normalité des données n'est pas rejetée, on peut donc l'admettre (faute de preuve contraire). On peut visualiser cela avec le Q-Q plot en faisant :

```
smi.qqplot(log_total_bill, line = "r")
```

Cela renvoie :



L'ajustement est bon.

On peut également analyser la normalité des données de `log_total_bill` suivant les modalités de `sex`.

Dans un premier temps, pour la modalité "Female", on fait :

```
log_total_bill_Female = np.log(tips.total_bill[tips.sex == "Female"])
scipy.stats.shapiro(log_total_bill_Female)
```

Cela renvoie :

```
ShapiroResult(statistic=0.9734253883361816, pvalue=0.07007473707199097)
```

On a p-valeur = 0.07007473707199097. Comme p-valeur > 0.05, la normalité des données est validée.

Puis, pour la modalité "Male", on fait :

```
log_total_bill_Male = np.log(tips.total_bill[tips.sex == "Male"])
scipy.stats.shapiro(log_total_bill_Male)
```

Cela renvoie :

```
ShapiroResult(statistic=0.9925491809844971, pvalue=0.5931006669998169)
```

On a p-valeur = 0.5931006669998169). Comme p-valeur > 0.05, la normalité des données est également validée.

On peut aussi étudier la normalité des données de `tip`. On fait :

```
scipy.stats.shapiro(tips.tip)
```

Cela renvoie :

```
ShapiroResult(statistic=0.897811233997345, pvalue=8.20057563521992e-12)
```

Ainsi, on a p-valeur = $8.20057563521992 \times 10^{-12}$, donc p-valeur < 0.001 ; le rejet de la normalité des données de `tip` est hautement significatif $\star\star\star$. On avait déjà remarqué cela avec la forme de l'histogramme associée.

En revanche, on peut voir si la distribution sous-jacente au logarithme de `log_tip`, caractère noté `log_tip` ; suit bien une loi normale. On fait :

```
log_tip = np.log(tips.tip)
scipy.stats.shapiro(log_tip)
```

Cela renvoie :

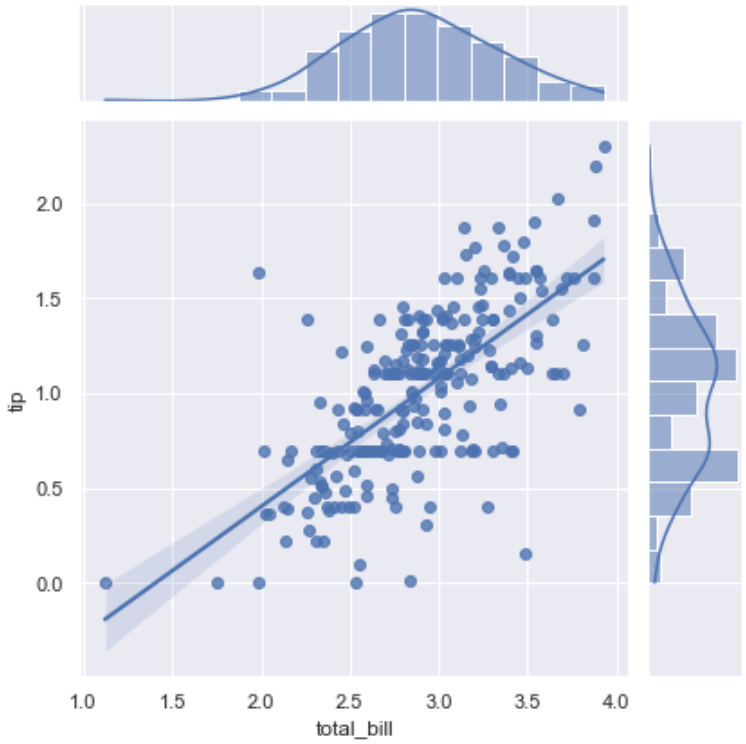
```
ShapiroResult(statistic=0.9888471961021423, pvalue=0.05621703341603279)
```

Ainsi, on a p-valeur = 0.05621703341603279, donc p-valeur > 0.05 (de justesse) ; la normalité des données n'est pas rejetée, on peut donc l'admettre.

Pour finir, par curiosité, on peut regarder si le lien linéaire entre `log_total_bill` et `log_tip` est plus flagrant que celui entre `total_bill` et `tip`. On fait :

```
sns.jointplot(x = log_total_bill, y = log_tip, kind = "reg")
```

Cela renvoie :



Le lien linéaire semble un peu meilleur avec les transformations logarithmiques.

5.2 Test d'une moyenne

Partant des données, on souhaite tester la conformité de la moyenne inconnue associée à un caractère quantitatif à une certaine norme Si la distribution sous-jacente du caractère est en adéquation avec la loi normale, alors on peut utiliser le test de Student. La commande clé est : `scipy.stats.ttest_1samp`.

Pour travailler sur un exemple, on considère le caractère quantitatif `log_total_bill` défini précédemment. On considère la moyenne de référence $\mu_0 = 2.95$ et on pose les hypothèses : $H_0 : \mu = 2.95$ contre $H_1 : \mu \neq 2.95$, où μ représente la moyenne inconnue associée à `log_total_bill`. Dès lors, on étudie si $\mu = 2.95$ en faisant :

```
scipy.stats.ttest_1samp(log_total_bill, popmean = 2.95)
```

Cela renvoie :

```
Ttest_1sampResult(statistic=-2.124481364663343, pvalue=0.034639873897269025)
```

Ainsi, on a p-valeur $= 0.034639873897269025$, donc p-valeur $\in]0.01, 0.05]$; on rejette H_0 et ce rejet est significatif $\star$. On peut donc affirmer, avec un faible risque de se tromper, que la moyenne inconnue associée à `log_total_bill` diffère de 2.95.

Un exemple de test unilatéral est le suivant. Si l'on considère les hypothèses : $H_0 : \mu = 2.81$ contre $H_1 : \mu > 2.81$, on fait :

```
scipy.stats.ttest_1samp(log_total_bill, popmean = 2.81, alternative = "greater")
```

Cela renvoie :

```
Ttest_1sampResult(statistic=2.858408457019687, pvalue=0.0023136007319686375)
```

Ainsi, on a p-valeur $= 0.0023136007319686375$, donc p-valeur $\in]0.001, 0.01]$; on rejette H_0 et ce rejet est très significatif $\star\star$. On peut donc affirmer, avec un faible risque de se tromper, que la moyenne inconnue associée à `log_total_bill` est supérieure à 2.81.

5.3 Test de comparaison de deux moyennes (échantillons indépendants)

Partant des données, on souhaite tester l'égalité de deux moyennes inconnues associées à deux caractères quantitatifs à partir de deux échantillons d'individus indépendants, et ainsi analyser l'homogénéité des sous-populations associées. Si les distributions sous-jacentes sont en adéquation avec des lois normales, on peut utiliser le test de Student à deux échantillons indépendants. La commande clé est : `scipy.stats.ttest_ind`.

Pour travailler sur un exemple, on considère les caractères quantitatifs `log_total_bill_Female` et `log_total_bill_Male` définis précédement. On pose les hypothèses : $H_0 : \mu_1 = \mu_2$ contre $H_1 : \mu_1 \neq \mu_2$, où μ_1 et μ_2 représentent les moyennes inconnues associées à `log_total_bill_Female` et `log_total_bill_Male`, respectivement. Pour mettre en œuvre le test de Student adéquat, on fait :
`scipy.stats.ttest_ind(log_total_bill_Female, log_total_bill_Male, equal_var = False)`

Cela renvoie :

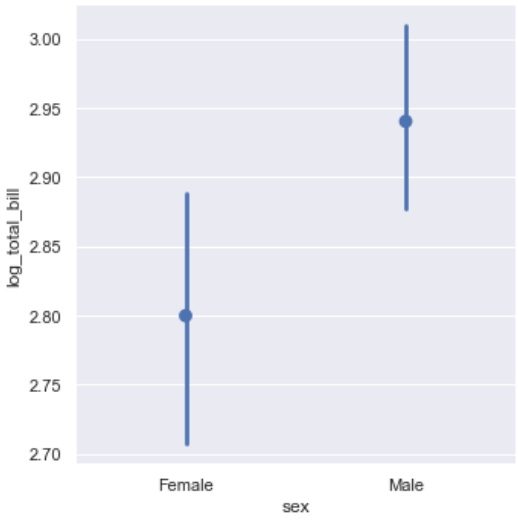
```
Ttest_indResult(statistic=-2.393945823575832, pvalue=0.01774056034844257)
```

Ainsi, on a p-valeur $= 0.01774056034844257$, donc p-valeur $\in]0.01, 0.05]$; on rejette H_0 et ce rejet est significatif $\star$. On peut donc affirmer, avec un faible risque de se tromper, que les moyennes inconnues associées à `log_total_bill_Female` et `log_total_bill_Male` diffèrent.

On peut visualiser cette différence en faisant :

```
df = pd.DataFrame({"log_total_bill" : log_total_bill, "sex" : tips.sex})
sns.catplot(x = "sex", y = "log_total_bill", data = df, kind = "point", join = False)
```

Cela renvoie :



Mais un test statistique était nécessaire pour confirmer la différence “statistique” de ces moyennes.

Un exemple de test unilatéral est le suivant. Si l'on considère les hypothèses : $H_0 : \mu_1 = \mu_2$ contre $H_1 : \mu_1 < \mu_2$, on fait :

```
scipy.stats.ttest_ind(log_total_bill_Female, log_total_bill_Male, equal_var = False,
alternative = "less")
```

Cela renvoie :

```
Ttest_indResult(statistic=-2.393945823575832, pvalue=0.008870280174221284)
```

Ainsi, on a p-valeur $= 0.008870280174221284$, donc p-valeur $\in]0.001, 0.01]$; on rejette H_0 et ce rejet est très significatif $\star\star$. On peut donc affirmer, avec un faible risque de se tromper, que la moyenne inconnue associée à `log_total_bill_Female` est strictement inférieure à celle associée à `log_total_bill_Male`.

Dans le cas d'échantillons appariés (un même échantillon d'individus sur lesquels on mesure un même caractère dans deux configurations différentes, créant ainsi deux listes de valeurs liés aux mêmes individus par paires de valeurs), on utilise la syntaxe : `scipy.stats.ttest_rel` au lieu de `scipy.stats.ttest_ind`. Le “rel” signifiant “related”, soit “lié” en français.

5.4 Test d'indépendance de deux caractères quantitatifs

Partant des données, on souhaite tester l'indépendance de deux caractères qualitatifs (avec des échantillons indépendants). On peut alors utiliser le test exact de Fisher, ou le test du Chi-deux d'indépendance. Les commandes clés sont : si les deux caractères ont deux modalités chacun, on peut utiliser le test de Fisher exact disponible dans `scipy` : `scipy.stats.fisher_exact`, sinon, on se tourne sur le test du Chi-deux d'indépendance : `scipy.stats.chi2_contingency`, respectivement.

Pour travailler sur un exemple, on considère les caractères qualitatifs `sex` et `smoker`. Les hypothèses sont : H_0 : "les caractères sont indépendants" contre H_1 : "les caractères sont dépendants". Pour mettre en œuvre le test exact de Fisher, on fait :

```
t = pd.crosstab(tips.sex, tips.smoker)
scipy.stats.fisher_exact(t)
```

Cela renvoie :

```
(1.0121836925960637, 1.0)
```

Dès lors, la p-valeur est donnée par la deuxième composante du vecteur. On a donc p-valeur = 1. Comme p-valeur > 0.05, on ne rejette pas l'hypothèse d'indépendance ; on peut l'admettre.

Avec le test du Chi-deux d'indépendance (plus exigeant en hypothèses et moins précis, mais employons le quand même), on fait :

```
t = pd.crosstab(tips.sex, tips.smoker)
scipy.stats.chi2_contingency(t)
```

Cela renvoie :

```
(0.0,
 1.0,
 1,
 array([[53.84016393, 33.15983607],
        [97.15983607, 59.84016393]]))
```

La p-valeur est donnée par la troisième composante du premier vecteur. On obtient ici aussi p-valeur = 1, et donc, la même conclusion.

5.5 Test de corrélation

Partant des données, on souhaite tester le lien linéaire entre deux caractères quantitatifs (avec des échantillons indépendants). Si les distributions sous-jacentes sont en adéquation avec des lois normales (pour faire simple), alors on peut utiliser le test de Pearson. La commande clé est : `scipy.stats.pearsonr`.

Pour travailler sur un exemple, on considère les caractères quantitatifs `log_total_bill` et `log_tip` déjà définis. Les hypothèses considérées sont : $H_0 : \rho = 0$ contre $H_1 : \rho \neq 0$, où ρ désigne le coefficient de corrélation de Pearson. Pour mettre en œuvre le test de Pearson, on fait :

```
scipy.stats.pearsonr(log_total_bill, log_tip)
```

Cela renvoie :

```
(0.6795702560274505, 2.0794565165370727e-34)
```

Ainsi, on a p-valeur $= 2.0794565165370727 \times 10^{-34}$, donc p-valeur < 0.001 ; on rejette H_0 et ce rejet est hautement significatif $\star\star\star$. On peut donc affirmer, avec un faible risque de se tromper, que le lien linéaire entre `log_total_bill` et `log_tip` est fort.

5.6 Test d'une proportion

Partant des données, on souhaite tester la valeur proportion inconnue associée à un caractère qualitatif binaire. Cette proportion inconnue correspond à la probabilité inconnue que le caractère soit égale à une de ces modalités (celles qui nous intéressent dans le contexte). Pour ce faire, on peut utiliser le test binomial. La commande clé est : `scipy.stats.binomtest`.

Pour travailler sur un exemple, on considère le caractère qualitatif `sex`. On aimerait affirmer, avec un faible risque de se tromper, que la proportion inconnue d'hommes payant l'addition dans le restaurant est strictement supérieure à 0.6

Les hypothèses considérées sont : $H_0 : p = 0.6$ contre $H_1 : p > 0.6$. Pour mettre en œuvre le test binomial, on fait :

```
scipy.stats.binomtest(157, n = 244, p = 0.6, alternative = "greater")
```

Cela renvoie :

```
BinomTestResult(k=157, n=244, alternative='greater',
proportion_estimate=0.6434426229508197, pvalue=0.09283443293725978)
```

Ainsi, on a p-valeur $= 0.09283443293725978$. Comme p-valeur > 0.05, on ne rejette pas H_0 ; les données ne nous permettent pas de conclure.

En moins précis mais plus populaire, on aurait pu utiliser le test de Wald. La commande clé est : `statsmodels.statsproportions_ztest`.

5.7 Test de comparaison de deux proportions (échantillons indépendants)

Partant des données, on souhaite tester l'égalité de deux proportions inconnues associées à deux caractères qualitatifs binaires. Pour ce faire, on peut utiliser le test exact de Fisher, ou le test de Wald, le premier étant plus précis que le deuxième, mais plus coûteux en temps de calcul (voire impossible à exécuter). Les commandes clés sont `scipy.stats.fisher_exact` et `statsmodels.statsproportions_ztest`.

Pour travailler sur un exemple, on considère le caractère qualitatif `sex`. On aimerait affirmer, avec un faible risque de se tromper, que la proportion inconnue d'hommes payant l'addition au dîner (`dinner`) dans le restaurant diffère de la proportion inconnue d'hommes payant l'addition au midi (`lunch`).

Les hypothèses considérées sont : $H_0 : p_1 = p_2$ contre $H_1 : p_1 \neq p_2$, où p_1 désigne la proportion inconnue d'homme payant l'addition au dîner, et p_2 la proportion inconnue d'hommes payant l'addition au midi. Pour mettre en œuvre le test exact de Fischer, on fait :

```
a = pd.crosstab(tips.sex[tips.time == "Dinner"], "freq")
a
b = pd.crosstab(tips.sex[tips.time == "Lunch"], "freq")
b
t = np.array([[124, 52], [33, 35]])
scipy.stats.fisher_exact(t)
```

Cela renvoie :

```
(2.5291375291375293, 0.0017373717860969416)
```

Ainsi, on a p-valeur = 0.0017373717860969416. Comme p-valeur $\in]0.001, 0.01]$; on rejette H_0 et ce rejet est très significatif $\star\star$. On peut donc affirmer, avec un faible risque de se tromper, que le la proportion inconnue d'homme payant l'addition au dîner diffère de celle payant l'addition au midi.

En moins précis mais plus populaire, on aurait pu utiliser le test de Wald. La commande clé est : `statsmodels.statsproportions_ztest`.

5.8 Test de positionnement

Le test de comparaison de moyenne avec échantillons indépendants nécéssite la normalité des données. Si cette normalité n'est pas validé, on peut tout de même comparer le positionnement numérique probable de deux caractères quantitatifs par le test de Mann-Whitney. La commande clé est `scipy.stats.mannwhitneyu`.

Pour travailler sur un exemple, on considère les caractères quantitatifs `total_bill` pour les hommes, et `total_bill` pour les femmes. On a déjà vu que la normalité est rejeté pour ces caractères. Comme le test de comparaison de moyenne standard n'est pas valable dans ce cas, on peut faire un test de positionnement. Les hypothèses considérées sont : H_0 : "il y a autant de chances qu'un caractère ait une valeur supérieure à l'autre, que vice-versa" contre H_1 : "un des deux caractères a plus de chances d'avoir une valeur supérieure à l'autre". Pour mettre en œuvre le test de Mann-Whitney, on fait :

```
scipy.stats.mannwhitneyu(tips.total_bill[tips.sex == "Male"],
tips.total_bill[tips.sex == "Female"], method = "exact")
```

Cela renvoie :

```
MannwhitneyuResult(statistic=8045.5, pvalue=0.021196582868044853)
```

Ainsi, on a p-valeur = 0.021196582868044853, donc p-valeur $\in]0.01, 0.05]$; on rejette H_0 et ce rejet est significatif $\star$. On peut donc affirmer, avec un faible risque de se tromper, que les caractères quantitatifs `total_bill` pour les hommes, et `total_bill` pour les femmes diffèrent quant à la probabilité de leur positionnement numérique.

5.9 Test de comparaison de plusieurs moyennes (ANOVA)

Partant des données, on souhaite tester l'égalité de plusieurs moyennes inconnues associées à plusieurs caractères quantitatifs à partir de deux échantillons d'individus indépendants, et ainsi analyser l'homogénéité des sous-populations associées. Si les distributions sous-jacentes sont en adéquation avec des lois normales, on peut utiliser le test d'analyse de la variance (ANOVA). La commande clé est : `sm.stats.anova_lm` ou `scipy.stats.f_oneway`.

Pour travailler sur un exemple, on considère le caractère quantitatif `log_total_bill` et le caractère qualitatif `day` définis précédement. On pose les hypothèses : H_0 : $\mu_1 = \mu_2 = \mu_3 = \mu_4$ contre H_1 : "au moins deux moyennes diffèrent", où μ_1, μ_2, μ_3 et μ_4 représentent les moyennes inconnues associées à `log_total_bill` pour les quatre modalités de `day` : `Sun`, `Sat`, `Thur` et `Fri`, respectivement. Pour mettre en œuvre le test ANOVA, on fait :

```
dataS = pd.DataFrame({"log_total_bill" : log_total_bill, "day" : tips.day})
model = sm.ols("log_total_bill~ C(day)", data = dataS).fit()
anova_table = smi.stats.anova_lm(model, typ=2)
anova_table
```

Cela renvoie :

```
              sum_sq     df       F    PR(>F)
C(day)      1.770736    3.0  3.1456  0.025847
Residual   45.033979  240.0     NaN       NaN
```

Ainsi, on a p-valeur = 0.025847, donc p-valeur $\in]0.01, 0.05]$; on rejette H_0 et ce rejet est significatif $\star$. On peut donc affirmer, avec un faible risque de se tromper, qu'au moins deux moyennes diffèrent. Autrement dit, il y a une influence du caractère `day` sur `log_total_bill`, et cette influence est significative.

On aurait aussi pu faire :

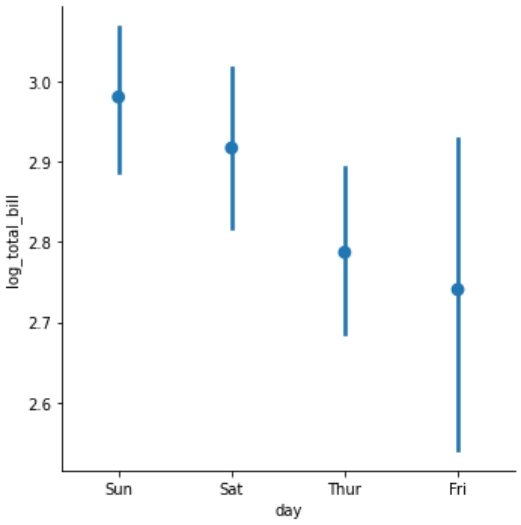
```
dataS = pd.DataFrame({"log_total_bill" : log_total_bill, "day" : tips.day})
scipy.stats.f_oneway(dataS["log_total_bill"][dataS["day"] == "Sun"],
                dataS["log_total_bill"][dataS["day"] == "Sat"],
                dataS["log_total_bill"][dataS["day"] == "Thur"],
                dataS["log_total_bill"][dataS["day"] == "Fri"])
```

Qui renvoie : `F_onewayResult(statistic=3.145600281643438, pvalue=0.025847292595926795)`. On retrouve alors p-valeur = 0.025847292595926795.

On peut visualiser cette différence de moyenne en faisant :

```
sns.catplot(x = "day", y = "log_total_bill", data = dataS,  kind = "point", join= False)
```

Cela renvoie :



La différence entre la première et la dernière moyenne est grande, maos un test était nécessaire pour valider statistiquement cette différence (les variances étant à prendre en compte).

Pour savoir quels sont les moyennes qui diffèrent le plus, on peut faire des tests "post-hoc", tels que les tests de Tukey (HSD) ou les tests de Bonferroni.

5.10 Intervalles de confiance

5.10.1 Intervalle de confiance pour une moyenne

Sous l'hypothèse de normalité des données, on rappelle qu'un intervalle de confiance d'une moyenne inconnue μ d'un caractère quantitatif au niveau $100(1-\alpha)\%$ est donné par la formule suivante :

$$i_\mu = \left[\overline{x} - t_\alpha(\nu)\frac{s}{\sqrt{n}}, \overline{x} + t_\alpha(\nu)\frac{s}{\sqrt{n}}\right],$$

où $\overline{x}$ est la moyenne des données, s est l'écart-type (corrigé) des données, et $t_\alpha(\nu)$ est le réel défini par

$$t_\alpha(\nu) = Q\left(1 - \frac{\alpha}{2}; \nu\right)$$

où $Q(x;\nu)$ désigne la fonction de quantile de la loi de Student à $\nu = n - 1$ degré de liberté. On peut coder cet intervalle de confiance en Python de la manière suivante :

```
def int_ech_moy(values, niveau = 0.95) :
    n = len(values)
    m = statistics.mean(values)
    s = statistics.variance(values)
    proba = 1 - (1 - niveau) / 2
```

```
ddl = n - 1
pp = scipy.stats.t.ppf(proba, ddl)
borneinf = m - pp *  np.sqrt(s / n)
bornesup = m + pp *  np.sqrt(s / n)
return(borneinf, bornesup)
```

Dès lors, par exemple, on peut avoir un intervalle de confiance pour la moyenne inconnue associée à `log_total_bill` au niveau 99% en faisant :

```
int_ech_moy(log_total_bill, 0.99)
```

Cela renvoie :

```
(2.817366696538782, 2.963253826211445
```

Ainsi, il y a 99% de chances que la moyenne inconnue associée à `log_total_bill` appartient à l'intervalle $(2.817366696538782, 2.963253826211445)$.

5.10.2 Intervalle de confiance pour une proportion

— On peut déterminer un intervalle de confiance exact d'une proportion inconnue p au niveau $100(1-\alpha)\%$. On peut alors utiliser l'intervalle de confiance binomial. La commande clé est : `scipy.stats.binomtest` avec l'option `proportion_ci`.
Dès lors, par exemple, on peut avoir un intervalle de confiance pour la proportion inconnue d'hommes payant l'addition dans le restaurant au niveau 95% en faisant :

```
result = scipy.stats.binomtest(157, n = 244)
ic = result.proportion_ci(confidence_level = 0.95)
ic
```

Cela renvoie :

```
ConfidenceInterval(low=0.5798344452011459, high=0.7035180842635822)
```

Ainsi, il y a 95% de chances que la proportion inconnue d'hommes payant l'addition dans le restaurant soit dans l'intervalle $(0.5798344452011459, 0.7035180842635822)$.

— Si la taille de l'échantillon n est très grand, un intervalle de confiance (approché et scolaire) d'une proportion inconnue p au niveau $100(1-\alpha)\%$ est donné par la formule suivante :

$$i_p = \left[f - z_\alpha \sqrt{\frac{f(1-f)}{n-1}}, f + z_\alpha \sqrt{\frac{f(1-f)}{n-1}} \right],$$

où f est la fréquence du caractère qualitatif binaire égale à la modalité considérée dans les données, et z_α est le réel défini par

$$z_\alpha = Q\left(1 - \frac{\alpha}{2}\right),$$

où $Q(x)$ désigne la fonction de quantile de la loi normale $\mathcal{N}(0,1)$. Celui-ci, appelé intervalle de confiance de Wald, est moins précis que l'intervalle de confiance binomiale On peut le coder en Python de la manière suivante :

```
def int_ech_prop(frequence, n, niveau = 0.95) :
    f = frequence
    proba = 1 - (1 - niveau) / 2
    pp = scipy.stats.norm.ppf(proba)
    borneinf = f - pp * np.sqrt(f * (1 - f) / (n - 1))
    bornesup = f + pp * np.sqrt(f * (1 - f) / (n - 1))
    return(borneinf, bornesup)
```

Dès lors, à titre d'exemple, on peut avoir un intervalle de confiance pour la proportion inconnue d'hommes payant l'addition dans le restaurant au niveau 95% en faisant :

```
int_ech_prop(157 / 244, 244, 0.95)
```

Cela renvoie :

```
(0.583219318660705, 0.7036659272409344)
```

Ainsi, il y a 95% de chances que la proportion inconnue d'hommes payant l'addition dans le restaurant soit dans l'intervalle $(0.583219318660705, 0.7036659272409344)$.

6 Élements de classification et de régression

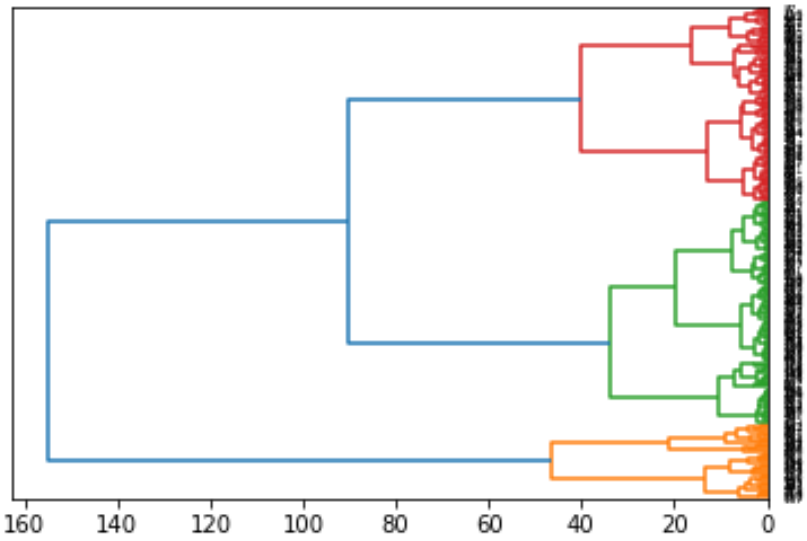
6.1 Classification

Partant des données, on souhaite désormais regrouper les individus qui se ressemblent ; c'est-à-dire, ceux qui ont des valeurs de caractères proches au sens de la distance mathématique. Pour ce faire, on peut utiliser l'algorithme de classification ascendante hiérarchique (CAH). La commande clé est `scipy.cluster.hierarchy.linkage`.

Pour travailler sur un exemple, on considère les caractères quantitatifs `total_bill`, `tips` et `size`. On aimerait faire trois groupes d'individus qui se ressemblent le plus quant à ces caractères. On met en oeuvre l'algorithme CAH (avec la méthode de Ward et la distance euclidienne) en faisant :

```
CAH = scipy.cluster.hierarchy.linkage(tips[["total_bill", "tip", "size"]],method = "ward",
metric = "euclidean")
scipy.cluster.hierarchy.dendrogram(CAH, labels = tips.index, orientation = "left",
color_threshold = 60)
```

Cela renvoie :



Les trois groupes d'individus obtenus se distinguent par les couleurs rouges, vert et orange (tous les individus sont présents sur l'axe ordonné à droite avec leur indices, lesquels sont illisibles car trop collés). Ils ont été obtenus après une coupure du dendrogramme au seuil de $t = 60$ (valeur approximative, $t = 71$, par exemple, aurait donné le même résultat, le but étant de couper le dendrogramme de sorte à avoir trois groupes). On peut identifier le groupe d'appartenance de chaque individu en faisant :

```
groupes_cah = scipy.cluster.hierarchy.fcluster(CAH, t = 60, criterion = "distance")
print(groupes_cah)
```

Cela renvoie :

```
[3 2 3 3 3 3 2 3 2 2 2 1 2 3 2 3 2 2 2 3 3 3 2 1 3 3 2 2 3 3 2 3 2 3 3 3 2
 2 3 1 2 3 2 2 1 3 3 1 3 3 2 2 1 2 3 3 1 3 2 1 3 2 2 3 2 3 2 2 3 2 2 2 3 3
 2 2 3 3 3 3 3 3 2 1 2 1 2 3 3 3 3 3 2 3 3 1 3 2 3 2 2 2 1 3 3 2 3 3 3 2 2
 2 1 3 3 3 1 2 2 3 2 2 2 2 2 1 2 2 2 3 3 3 2 2 3 2 2 2 2 2 3 1 1 3 2 2 3 2
 2 2 2 2 2 3 3 1 1 3 2 2 3 2 2 2 3 3 3 1 2 2 1 2 2 1 3 1 3 2 2 1 1 3 1 3 1
 3 3 1 3 3 2 3 3 2 3 2 2 1 2 2 3 2 2 2 3 2 3 1 3 2 1 3 1 2 1 2 3 2 2 1 2 2
 2 2 2 2 2 3 2 3 3 2 2 2 2 2 2 1 1 1 3 3 3 3]
```

Ainsi, par exemple, le premier individu appartient au groupe labellisé `3` (le Groupe 3), le deuxième individu appartient au groupe labellisé `2` (le Groupe 2), et ainsi de suite.

Une fois cette classification faite, on peut s'intéresser au parangon (individu de chaque groupe qui est le plus représentatif de son groupe en termes de caractéristiques), le caractère dominant dans chaque groupe, ou le caractère qui a été le plus discriminant dans les regroupements.

Pour aller plus loin dans les méthodes de classification, voir le module `Scikit_learn` qui nous permet, entre autres, d'utiliser la méthodes des k-means, faire des arbres de décision, etc.

6.2 Régression linéaire

Cette partie présente les bases de la régression linéaire.

6.2.1 Régression linéaire simple

On souhaite expliquer un caractère quantitatif en fonction d'un autre caractère quantitatif. Expliquer au sens "comportement numérique" du terme. On dispose d'observations de ces deux caractères, soit deux séries de valeurs, constituant ainsi les données. Si le nuage de points associé laisse présager un lien linéaire entre ces deux caractères, on peut utiliser le modèle de régression linéaire simple. Ainsi, en notant Y le caractère à expliquer et X le caractère explicatif, le modèle de régression linéaire simple est de la forme suivante :

$$Y = \beta_0 + \beta_1 X + \epsilon,$$

où β_0 et β_1 sont des coefficients inconnus qu'il faudra estimer avec les données, et ϵ est une variable d'erreur qui modélise une somme de petites erreurs aléatoires provenant de l'expérimentation d'où provient les données.

Par la méthode des moindres carrés ordinaires, on peut estimer β_0 et β_1 à l'aide des données, donnant ainsi les estimations b_0 et b_1, respectivement. Dès lors, pour une valeur x de X donnée, une estimation de la valeur moyenne de Y, notée y, est donnée par l'équation :

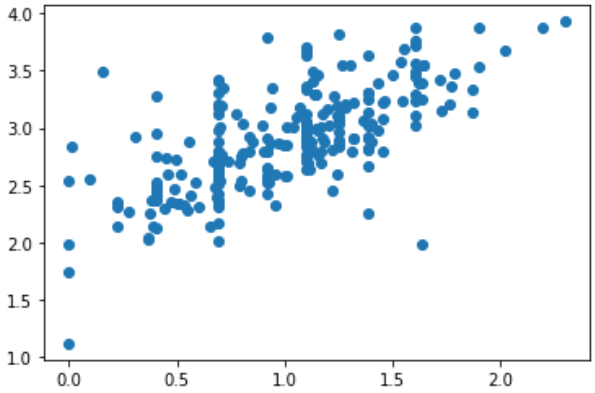
$$y = b_0 + b_1 x.$$

Aussi, avec x variable, la droite d'équation $y = b_0 + b_1 x$ est appelée droite de régression ; c'est cette droite qui fournit un bon ajustement du nuage de points associés.

Pour travailler sur un exemple, on considère les caractères quantitatifs `log_total_bill` et `log_tips` (lesquels ont des distributions sous-jacentes en adéquation avec la loi normale, ce qui est préférable avec le modèle de régression linéaire). On souhaite expliquer `log_total_bill` en fonction de `log_tips`. Pour avoir le nuage de points associé, on fait :

```
plt.scatter(log_tip, log_total_bill)
```

Cela renvoie :



Vu la forme allongée du nuage, un lien linéaire entre `log_total_bill` et `log_tips` est envisageable. On met en œuvre le modèle de régression linéaire simple en faisant :

```
X = smi.add_constant(log_tip)
model = smi.OLS(log_total_bill, X)
results = model.fit()
print(results.summary())
```

Cela renvoie :

```
                            OLS Regression Results
==============================================================================
Dep. Variable:             total_bill   R-squared:                       0.462
Model:                            OLS   Adj. R-squared:                  0.460
Method:                 Least Squares   F-statistic:                     207.7
Date:                Sun, 05 Jun 2022   Prob (F-statistic):           2.08e-34
Time:                        21:15:21   Log-Likelihood:                -69.191
No. Observations:                 244   AIC:                             142.4
Df Residuals:                     242   BIC:                             149.4
Df Model:                           1
Covariance Type:            nonrobust
```

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.2048      0.052     42.512      0.000       2.103       2.307
tip            0.6838      0.047     14.410      0.000       0.590       0.777
==============================================================================
Omnibus:                       15.967   Durbin-Watson:                   1.944
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               40.213
Skew:                           0.195   Prob(JB):                     1.85e-09
Kurtosis:                       4.950   Cond. No.                         4.84
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

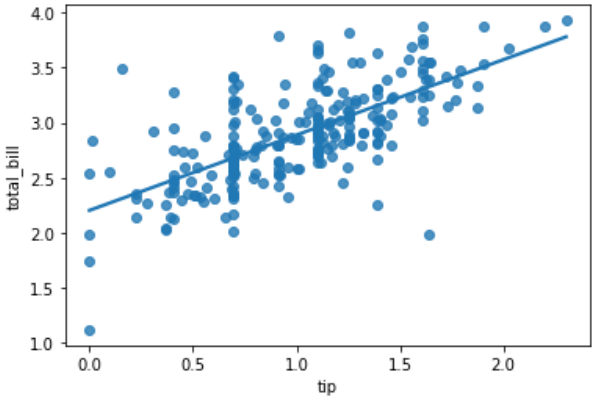
On a beaucoup d'informations. Entre autres, on a R-squared $= 0.462$, qui est assez éloigné de 1, ce qui signifie que le modèle est plutôt moyen en terme prédictif. La p-valeur associée à β_1 est P>|t| 0.000, ce qui signifie que p-valeur < 0.001, et que `log_tips` à une influence hautement significative sur `log_total_bill`. On a également les valeurs de b_0 et b_1, avec $b_0 = 2.2048$ et $b_1 = 0.6838$, d'où l'équation de la droite de régression :

$$y = 2.2048 + 0.6838x.$$

On peut visualiser cette droite en faisant :

```
sns.regplot(x = log_tip, y = log_total_bill, ci = None)
```

Cela renvoie :



Ainsi, pour la valeur 2.5 de `log_tips` par exemple, une valeur moyenne de `log_total_bill` s'obtient en faisant :

```
results.predict((1,2))
```

Cela renvoie : `array([3.9142754])`, soit 3.9142754, ce qui correspond à $2.2048 + 0.6838 \times 2.5 = 3.9143$.

6.2.2 Régression linéaire multiple

Le cadre est similaire à celui de la régression linéaire simple, sauf qu'au lieu d'un seul caractère explicatif, on en a plusieurs. On dispose d'observations de ces caractères, soit plusieurs séries de valeurs, constituant ainsi les données. Si les nuages de points associés au caractère que l'on souhaite expliquer laisse présager un lien linéaire entre ces deux caractères, on peut penser à utiliser le modèle de régression linéaire multiple. Ainsi, en notant Y le caractère à expliquer et $X_1, \ldots, X_p$ les caractères explicatifs, le modèle de régression linéaire multiple s'écrit sous la forme :

$$Y = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \epsilon,$$

où $\beta_0, \beta_1 \ldots, \beta_p$ sont des coefficients inconnus qu'il faudra estimer avec les données, et ϵ est une variable d'erreur.

Par la méthode des moindres carrés ordinaires, on peut estimer $\beta_0, \beta_1 \ldots, \beta_p$ à l'aide des données, donnant ainsi les estimations $b_0, b_1 \ldots, b_p$, respectivement. Dès lors, pour des valeurs $x_1, x_2, \ldots, x_p$ de $X_1, X_2, \ldots, X_p$ données, une estimation de la valeur moyenne de Y, notée y, est donnée par l'équation :

$$y = b_0 + b_1 x_1 + \ldots + b_p x_p.$$

Pour travailler sur un exemple, on considère les caractère quantitatifs `log_total_bill`, `log_tips` et `size`. On souhaite expliquer `log_total_bill` en fonction de `log_tips` et `size`. On met en œuvre le modèle de régression linéaire multiple en faisant :

```
X = pd.DataFrame({"const" : 1, "log_tip" : log_tip, "size" : tips.size})
model = smi.OLS(log_total_bill, X)
results = model.fit()
print(results.summary())
```

Cela renvoie :

```
                            OLS Regression Results
==============================================================================
Dep. Variable:             total_bill   R-squared:                       0.462
Model:                            OLS   Adj. R-squared:                  0.460
Method:                 Least Squares   F-statistic:                     207.7
Date:                Mon, 06 Jun 2022   Prob (F-statistic):           2.08e-34
Time:                        01:23:49   Log-Likelihood:                -69.191
```

```
No. Observations:                 244   AIC:                             142.4
Df Residuals:                     242   BIC:                             149.4
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       7.558e-07   1.78e-08     42.512      0.000    7.21e-07    7.91e-07
log_tip        0.6838      0.047     14.410      0.000       0.590       0.777
size           0.0013   3.04e-05     42.512      0.000       0.001       0.001
==============================================================================
Omnibus:                       15.967   Durbin-Watson:                   1.944
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               40.213
Skew:                           0.195   Prob(JB):                     1.85e-09
Kurtosis:                       4.950   Cond. No.                     9.51e+18
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The smallest eigenvalue is 7.88e-30. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.
```

On a beaucoup d'informations. Entre autres, on a R-squared = 0.462, qui est assez éloigné de 1, ce qui signifie que le modèle est plutôt moyen en terme prédictif. Au vu des p-valeurs, `log_tips` et `tips.size` ont des influences hautement significatives sur `log_total_bill`. On a également les valeurs de b_0, b_1 et b_2, avec $b_0 = 7.558 \times 10^{-7}$, $b_1 = 0.6838$ et $b_2 = 0.0013$, ce qui nous permet de faire des prédictions sur la valeur moyenne de `log_total_bill`. Ainsi, pour la valeur de 2.5 pour `log_tips` et la valeur de 3 pour `tips.size` par exemple, une valeur moyenne de `log_total_bill` s'obtient en faisant :

```
results.predict((1,2.5, 3))
```

Cela renvoie : `array([1.713374])`, soit 1.713374.

Pour aller plus loins dans cette direction, voir :

—

6.3 Régression logistique simple

On souhaite expliquer un caractère qualitatif binaire en fonction d'un caractère quantitatif. On dispose d'observations de ces caractères, soit deux séries de valeurs, constituant ainsi les données.

Dès lors, on peut utiliser le modèle de régression logistique simple. Ainsi, en notant Y le caractère à expliquer et X le caractère explicatif, le modèle de régression logistique simple s'écrit sous la forme :

$$p(x) = \frac{\exp(\beta_0 + \beta_1 x)}{1 + \exp(\beta_0 + \beta_1 x)},$$

où $p(x)$ est la probabilité que Y soit égale à une modalité de référence, codée par 1 en toute circonstance, l'autre par 0 sachant que X vaut x, et β_0 et β_1 sont des coefficients inconnus qu'il faudra estimer avec les données.

Par la méthode du maximum de vraisemblance, on peut estimer β_0 et β_1 à l'aide des données, donnant ainsi les estimations b_0 et b_1, respectivement. Dès lors, pour une valeur x de X donnée, une estimation de $p(x)$ est est donnée par l'équation :

$$p_*(x) = \frac{\exp(b_0 + b_1 x)}{1 + \exp(b_0 + b_1 x)},$$

On peut alors faire des prédictions sur Y : si $p_*(x)$ dépasse 0.5, alors Y a plus de chances d'être égale à 1 (codant la modalité de référence), qu'à 0 (codant l'autre modalité).

Pour travailler sur un exemple, on considère le caractère qualitatif `sex` et le caractère quantitatif `log_total_bill`. On souhaite expliquer `sex` en fonction de `log_totall_bill`. On met en œuvre le modèle de régression logistique simple en faisant :

```
tips["sex"] = tips.sex.factorize()[0] # Codant ainsi ‘‘Female’’ en 0, et ‘‘Male’’ en 1
X = smi.add_constant(log_total_bill)
logisticReg = smi.Logit(tips.sex, X).fit()
print(logisticReg.summary())
```

Cela renvoie :

```
Optimization terminated successfully.
         Current function value: 0.639495
         Iterations 5
                           Logit Regression Results
==============================================================================
Dep. Variable:                    sex   No. Observations:                  244
Model:                          Logit   Df Residuals:                      242
Method:                           MLE   Df Model:                            1
Date:                Mon, 06 Jun 2022   Pseudo R-squ.:                 0.01829
Time:                        02:10:55   Log-Likelihood:                -156.04
converged:                       True   LL-Null:                       -158.94
Covariance Type:            nonrobust   LLR p-value:                   0.01589
==============================================================================
```

```
                coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          -1.5601      0.914     -1.708      0.088      -3.351       0.231
total_bill      0.7491      0.317      2.365      0.018       0.128       1.370
==============================================================================
```

Au vu des p-valeurs, notamment celle associée à β_1, `log_totall_bill` a une influence significative sur `sex`. De plus, les coefficients β_0 et β_1 ont été estimés par $b_0 = -1.5601$ et $b_1 = 0.7491$, respectivement.

On peut alors faire des prédictions en faisant :

```
logisticReg.predict((1,2.5))
```

Cela renvoie : `array([0.57755338])`. Ainsi, on a $p_*(x) = 0.57755338 > 0.5$ (de justesse), donc un individu vérifiant `log_total_bill` égal à 2.5 a plus de chances d'être un homme qu'une femme.

On peut mettre en œuvre la régression logistique multiple en adaptant les formules et commandes précédentes. Pour allez plus loin, on peut s'intéresser à la matrice de confusion (à partir d'un jeu de données d'entraînement et d'un jeu de données test ou pas), analyser la performance du modèle avec la courbe ROC, des pseudo R^2, etc.