TD Info 2 : Structures de données - Partie 2
Proposition de correction

Contents

Durant cette séance, nous allons poursuivre notre étude de différentes structures de données ("objet" contenant des données) : les dictionnaires et les tableaux.

1 Les dictionnaires

1.1 Intérêt d'un dictionnaire

On dispose du texte de la Petite Sirène, en anglais, dans lequel on a enlevé les points et tout autre signe de ponctuation. Le texte anglais présente l'intérêt de ne pas utiliser d'accents ni d'apostrophes. Procédons à l'ouverture du fichier contenant le texte :

[2]:
```
# Ouverture du fichier en mode lecture ('r' pour read), on ne pourra pas le modifier :
fichier = open("littleMermaid.txt",'r')

# On place le contenu du fichier dans une liste en lisant les lignes :
Texte = fichier.readlines()

# On ferme le fichier, inutile désormais
fichier.close()
```

[3]:
```
len(Texte)
```

[3]: 1

Pour l'instant la liste `Texte` ne contient qu'un seul élément, qui est la chaîne de caractères de l'ensemble du texte.

On forme alors la liste contenant les mots “séparés”, en s’assurant tout d’abord de transformer toutes les majuscules en minuscules :

[4]:
```
# On extrait le texte (seule élément de la liste Texte), et modifie les majuscules en␣
 ↪minuscules avec lower()
Texte = Texte[0].lower()

# La commande 'split' permet de séparer ou couper une chaine de caractères à chaque␣
 ↪argument (ici un espace)
Mots = Texte.split(' ')

# On supprime quelques caractères vides qui traînent...
Mots.remove("")
```

[6]:
```
len(Mots)
```

[6]: 9199

Q1. Afficher les 30 premiers mots du texte.

[13]:
```
print(Mots[:30])
```

```
['far', 'out', 'in', 'the', 'wide', 'seawhere', 'the', 'water', 'is', 'blue',
'as', 'the', 'loveliest', 'cornflower', 'and', 'clear', 'as', 'the', 'purest',
'crystal', 'where', 'it', 'is', 'so', 'deep', 'that', 'very', 'very', 'many',
'churchtowers']
```

On souhaiterai faire une analyse statistique de ce texte : par exemple, on peut s’intéresser à déterminer quel mot apparait le plus grand nombre de fois. Ou bien déterminer si un mot apparait dans le texte, et si oui, combien de fois.

Pour cela, il va falloir effectuer un décompte de l’occurence (nombre d’apparitions) de chaque mot dans le texte : il va nous falloir lier ou associer un mot à un nombre. C’est pourquoi nous allons introduire une nouvelle structure de données : les dictionnaires.

1.2 Création et utilisation de dictionnaires

Un dictionnaire présente de nombreuses similitudes avec les listes, mais intéressons-nous à ce qui diffère : pour une liste`L`, on accéde aux éléments individuels par le biais d’un indice. Ainsi on accède à l’élément en position `i`,nommé indice, via `L[i]`. En voici un exemple :

[8]:
```
Newton = ["Tout corps persévère dans l'état de repos ou de mouvement uniforme en ligne␣
 ↪droite dans lequel il se trouve, à moins que quelque force n'agisse sur lui, et ne␣
 ↪le contraigne à changer d'état","Les changements qui arrivent dans le mouvement sont␣
 ↪proportionnels à la force motrice ; et se font dans la ligne droite dans laquelle␣
 ↪cette force a été imprimée","L'action est toujours égale à la réaction ;␣
 ↪c'est-à-dire que les actions de deux corps l'un sur l'autre sont toujours égales et␣
 ↪de sens contraires"]
```

Dans la liste `Newton` se trouvent les énoncés originels des trois lois de Newton. Pour accéder à la deuxième lois de Newton :

[9]:
```
Newton[1]
```

[9]:
```
'Les changements qui arrivent dans le mouvement sont proportionnels à la force
motrice ; et se font dans la ligne droite dans laquelle cette force a été
imprimée'
```

Pour donner plus de “sens” à la structuration de ces données, on peut utiliser un dictionnaire. Dans ce cas, on accède à un élément individuel via une *clef*, et non un indice. Cette clef est une variable, donc peut être un nombre, une chaîne de caractère, … Voici une exemple de création d'un dictionnaire :

[14]:
```
Lois_Newton = {"1ere loi" : "Tout corps persévère dans l'état de repos ou de mouvement␣
↪uniforme en ligne droite dans lequel il se trouve, à moins que quelque force␣
↪n'agisse sur lui, et ne le contraigne à changer d'état",
               "2eme loi" : "Les changements qui arrivent dans le mouvement sont␣
↪proportionnels à la force motrice ; et se font dans la ligne droite dans laquelle␣
↪cette force a été imprimée",
               "3eme loi" : "L'action est toujours égale à la réaction ; c'est-à-dire␣
↪que les actions de deux corps l'un sur l'autre sont toujours égales et de sens␣
↪contraires"}
```

A chaque élément (ici une loi de Newton), on a associé son *nom* ou *appélation*, qui est ici la clef du dictionnaire. Pour accéder à la 2ème loi de Newton :

[15]:
```
Lois_Newton["2eme loi"]
```

[15]:
```
'Les changements qui arrivent dans le mouvement sont proportionnels à la force
motrice ; et se font dans la ligne droite dans laquelle cette force a été
imprimée'
```

Cette structure de données est au final semblable au classiques “dictionnaires” de langues. Mais contrairement à ceux-ci, ils ne sont pas triés ou ordonnés.

Remarque : On peut voir une liste comme un dictionnaire dont les clefs sont les entiers commençant à 0.

Autre exemple de création d'un dictionnaire :

[12]:
```
Stocks = dict()
Stocks["oranges"] = 18
Stocks["pommes"] = 3
Stocks["kiwis"] = 41
Stocks
```

[12]:
```
{'oranges': 18, 'pommes': 3, 'kiwis': 41}
```

Puis une modification :

[13]:
```
Stocks["kiwis"] -= 5
Stocks
```

[13]:
```
{'oranges': 18, 'pommes': 3, 'kiwis': 36}
```

Enfin il faut bien distinguer la façon d'obtenir une clef, ou l'élément associé à la clef :

[14]:
```
for element in Stocks :
    print(element)
    print(Stocks[element])
```

```
oranges
18
pommes
3
kiwis
36
```

Pour tester si une clef existe :

[15]:
```
print("kiwis" in Stocks)
print("mandarines" in Stocks)
```

```
True
False
```

De nombreuses fonctions existent pour les dictionnaires, dont un certain nombre similaires à celles pour les listes (`len`, `copy`).

Q2. Créer un dictionnaire dont les clefs sont vos matières en TPC, et les valeurs associées le nombre d'heures hebdomadaires.

[8]:
```
Matières = dict()
Matières["Maths"] = 10
Matières["Physique"] = 7.5
Matières["Chimie"] = 5.5

# Ou bien :
Matières = {"Maths" : 10,
            "Physique" : 7.5,
            "Chimie" : 5.5}
```

[9]:
```
print(Matières)
```

```
{'Maths': 10, 'Physique': 7.5, 'Chimie': 5.5}
```

1.3 Exercices

Q3. Créer le dictionnaire `Mots_occ` dont les clefs sont les mots apparaissants dans le texte de la petite sirène, et les éléments associés leur nombre d'occurences.

[10]:
```
Mots_occ = dict() # création du dictionnaire (clefs : mots du texte, auquels on␣
 ↪associe leur occurence)

for mot in Mots : # parcourt de la liste des mots du texte
    if mot in Mots_occ : # si le mot est déjà dans le dictionnaire
        Mots_occ[mot] += 1 # on ajoute 1 à son occurence
    else :
        Mots_occ[mot] = 1 # sinon on créé l'entrée dnas le dictionnaire, avec une␣
 ↪occurence de 1
```

[11]:
```
Mots_occ["mermaid"], Mots_occ["sea"]
```

[11]:
```
(37, 39)
```

Q4. Créer le dictionnaire `Occ_mots` dont les clefs sont les occurences, et les éléments associés les listes de mots qui ont cette occurence dans le texte de la petite sirène.

[12]:
```
Occ_mots = dict()

for mot in Mots_occ :
    if Mots_occ[mot] in Occ_mots :
        Occ_mots[Mots_occ[mot]].append(mot)
    else :
        Occ_mots[Mots_occ[mot]]=[mot]
```

[19]:
```
Occ_mots[12]
```

[19]:
```
['far',
 'beings',
 'great',
 'high',
 'those',
 'day',
 'stood',
 'immortal',
 'heart']
```

Q5. Faire un graphique représentant la fréquence normalisée de toutes les lettres de l'alphabet dans le texte de la petite sirène.

[20]:
```
Lettres_occ = dict()

cpt = 0

for lettre in Texte : # parcourt du texte (chaine de caractère)
    if lettre != " ": # on ne tient pas compte des espaces
        if lettre in Lettres_occ : # si la lettre est déjà dans le dictionnaire
            Lettres_occ[lettre] += 1 # on ajoute 1 à son occurence
        else :
            Lettres_occ[lettre] = 1 # sinon on créé l'entrée dans le dictionnaire,␣
 ↪avec une occurence de 1
        cpt += 1

Lettres_freq = dict()

for lettre in  Lettres_occ :
    Lettres_freq[lettre] = Lettres_occ[lettre] / cpt *100
```

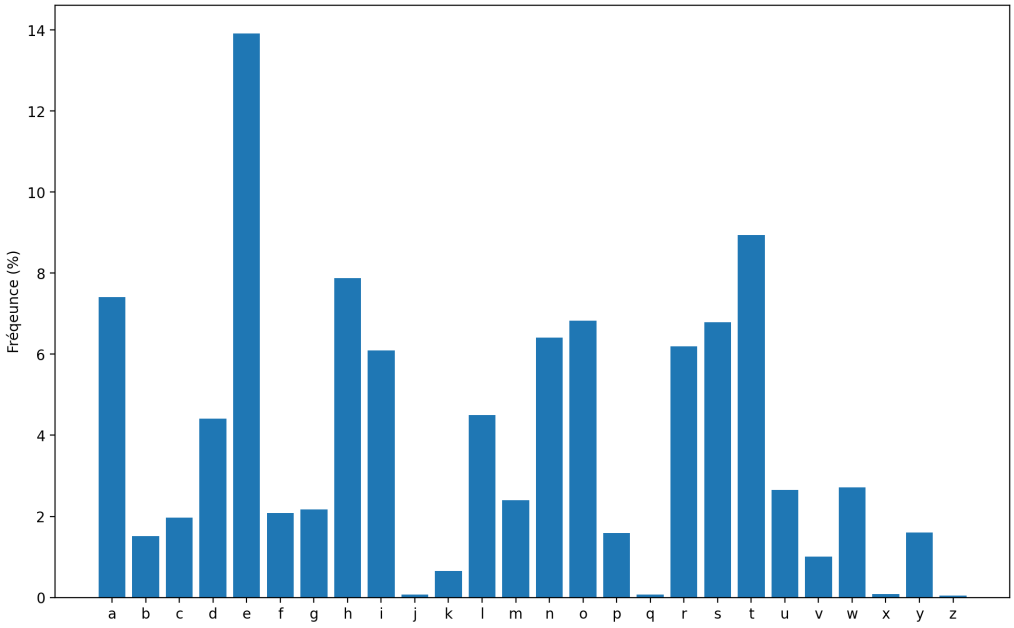
[21]:
```
Lettres_freq
```

[21]:
```
{'f': 2.0830690381346257,
 'a': 7.40364854235912,
 'r': 6.190850734529216,
 'o': 6.832770913150483,
 'u': 2.648872199528075,
```

```
't': 8.943749524268641,
'i': 6.09697307994824,
'n': 6.403978382767107,
'h': 7.883185750894375,
'e': 13.919265217060362,
'w': 2.7173775150331108,
'd': 4.404638063583081,
's': 6.789637936721387,
'b': 1.512191408925989,
'l': 4.503590185979245,
'v': 1.0072818613147947,
'c': 1.9688935122928983,
'p': 1.5883084261538072,
'y': 1.6009945956917768,
'm': 2.3951488087686803,
'g': 2.1769466927156014,
'j': 0.07357978332022429,
'k': 0.6520691142516428,
'x': 0.08372871895060005,
'q': 0.06596808159744247,
'z': 0.05074467815187882,
'\n': 0.0025372339075939413}
```

[22]:
```
Alphabet = "abcdefghijklmnopqrstuvwxyz"
Occ = [Lettres_freq[lettre] for lettre in Alphabet]
X = [k for k in range(len(Alphabet))]

ratio, dpi =  1.5, 200
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.bar(X, Occ)
plt.ylabel("Fréqeunce (%)")
plt.xticks(X, [lettre for lettre in Alphabet])
plt.show()
```



2 Les tableaux

2.1 Qu'est-ce qu'un tableau ?

Un tableau est un concept informatique théorique, qui permet de structurer de nombreuses données. Il peut d'agir d'un tableau 2d; contenant des variables (nombres, chaines de caratères, ...) disposées en lignes et colonnes (tout comme ce que désigne un tableau dans le vocabulaire courant, analogue aux matrices en maths), ou 1d (analogue aux vecteurs en maths), ou même de dimensions $d > 3$! Ils possèdent de nombreuses implémentations dans les différents langages informatiques.

On se restreindra ici aux tableau 1d et 2d, dans le langage python.

2.2 Réalisation de tableaux en python

Nous avons déjà, à de nombreuses reprises, utilisé des listes python, qui est un exemple de tableau 1d :

[29]:
```
L = [i**2 for i in range(10)]
print(L)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

L'affichage de ce tableau est fait de façon horizontal, c'est un choix (qui est naturel pour des questions de place occupée...), mais il pourrait tout aussi bien être vertical, comme on le fait généralement en maths pour les vecteurs.

Avec des listes de listes, on peut créer des tableaux 2d :

[23]:
```
Tab = [[6, 3, 4], [5, 10, 6], [7, 10, 5], [8, 5, 6]]
print(Tab)
```

```
[[6, 3, 4], [5, 10, 6], [7, 10, 5], [8, 5, 6]]
```

On représente généralement ce tableau en suivant la convention suivante :

$$\begin{matrix} 6 & 3 & 4 \\ 5 & 10 & 6 \\ 7 & 10 & 5 \\ 8 & 5 & 6 \end{matrix}$$

Cela signifie que notre liste de liste est une liste des lignes du tableau.

Q6. Qu'affiche `Tab[2]` ?

[44]:
```
Tab[2]
```

[44]:
```
[7, 10, 5]
```

`Tab[2]` est la 3ème ligne du tableau (début de la numérotation des lignes à 0).

Q7. Modifier le nombre 8, à la quatrième ligne et première colonne, par 3.

[25]:
```
Tab[3][0] = 3
Tab
```

[25]:
```
[[6, 3, 4], [5, 10, 6], [7, 10, 5], [3, 5, 6]]
```

Pour accéder à un élément particulier, on utilisera `Tab[i][j]` avec `i` le numéro de la ligne et `j` de la colonne selon la représentation usuelle (numérotation des lignes et colonnes commençant à 0).

Lorsqu'on souhaite utiliser un talbeau pour stocker des données numériques (nombres flottants), on utilise généralement et préférentiellement un tableau numpy. Bien qu'absents du programme, ces tableaux sont employés (jusqu'à présent) dans la majorité des épreuves de modélisation du concours : il n'est donc pas inutile d'en découvrir quelques propriétés !

Pour les utiliser, il faut déjà importer la bibliothèque *numpy*, de façon usuelle avec l'alias *np* :

[2]:
```
import numpy as np
```

On peut alors créer des tableaux 1d ou 2d :

[24]:
```
# Tableau 1d
a = np.array([1, 2, 3])
print(a)
# Tableau 2d
b = np.array([[1,2,3],[4,5,6]])
print(b)
```

```
[1 2 3]
[[1 2 3]
 [4 5 6]]
```

Pour l'instant, peu de différences avec les listes de liste, ormis le fait que pour accéder (et/ou modifier) l'élément à la ligne i et à la colonne j, on utilise la sytntaxe `b[i,j]` et non `b[i],[j]`.

[18]:
```
print(b[0, 0], b[0, 1], b[1, 0])
```

```
1 2 4
```

On peut aussi créer aisément des tableaux usuels :

[26]:
```
# tableau rempli de 0, de 2 lignes et 3 colonnes :
a = np.zeros((2,3))
print(a)

# tableau rempli de 1, de 2 lignes et 3 colonnes :
b = np.ones((3,2))
print(b)

# matrice identité de dimension 3 :
d = np.eye(3)
print(d)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1.]
 [1. 1.]
 [1. 1.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Q8. Extraire du tableau suivant `a` un tableau carré

$$\begin{matrix} 2 & 3 \\ 6 & 7 \end{matrix}$$

[51]:
```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
b = a[0:2, 1:3] # on extrait la partie comprenant les lignes 0 et 1, et les colonnes 1␣
 ↪et 2
print(b)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
[[2 3]
 [6 7]]
```

Attention : comme pour le listes, `b` se réfère au même emplacement mémoire que `a`, ce qui veut dire qu'en modifiant `b`, on modifie aussi `a` :

[53]:
```
print(a[0, 1])
b[0, 0] = 77
print(a[0, 1])
```

```
77
77
```

L'intérêt de la bibliothèque `numpy` est le fait qu'elle est construite pour effectuer des opérations mathématiques, selon le modèle des vecteurs et matrices. Par exemple, des opérations sur des tableaux (1d, 2d, …) :

[27]:
```
print("Addition de listes [1,2,3]+[4,5,6] :",[1,2,3]+[4,5,6])
print("Addition de tableaux [1,2,3]+[4,5,6] :",np.array([1,2,3])+np.array([4,5,6]))
print("Multiplication de tableaux [1,2,3]*[4,5,6] :",np.array([1,2,3])*np.
  ↪array([4,5,6]))
```

```
Addition de listes [1,2,3]+[4,5,6] : [1, 2, 3, 4, 5, 6]
Addition de tableaux [1,2,3]+[4,5,6] : [5 7 9]
Multiplication de tableaux [1,2,3]*[4,5,6] : [ 4 10 18]
```

On peut aussi appliquer directement des fonctions aux tableaux :

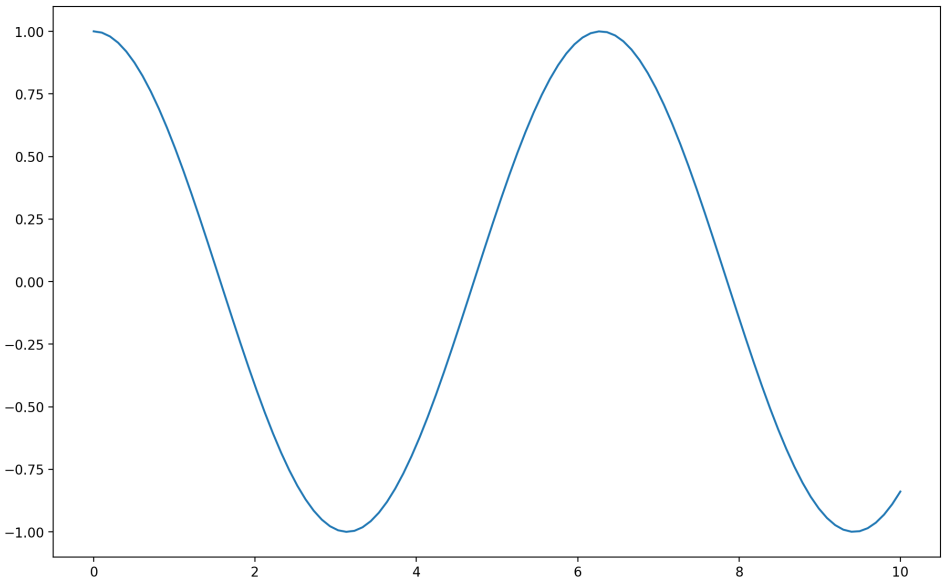
[55]:
```
print("cos appliquée à un tableau 1d :",np.cos([0,0.1,0.2,0.3]))
```

```
cos appliquée à un tableau 1d : [1.         0.99500417 0.98006658 0.95533649]
```

Ce qui est très pratique :

[28]:
```
ratio, dpi =  1.5, 200
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)

X = np.linspace(0,10,100)
Y = np.cos(X)
plt.plot(X,Y)
plt.show()
```



2.3 Manipulation de tableaux : visualisation à l'aide d'une image

2.3.1 Une image comme tableau

Une image en "niveau de gris" peut-être vue comme un tableau 2d de nombres. La fonction `imread` de matplotlib permet de transformer une image en"niveau de gris" au format png en un tel tableau :

[3]:
```
Image = plt.imread("im/Boltzmann.png")
ratio, dpi =  1.5, 70
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.gray()
```

```
plt.imshow(Image)
plt.show()
```

Cette image est un portrait de Ludwig Boltzmann, père de la vision statistique de la physique moderne. Elle est constituée d'un certain nombres de pixels (**PI**x**EL** pour **PI**cture **EL**ement) :

[68]:
```
print(Image.shape)
```

```
(268, 211)
```

Cette image est donc constituée de 268 lignes et 211 colonnes, ce qui donne 56 548 pixels. Regardons plus en détails ce que contient la variable Image :

[80]:
```
Image
```

[80]:
```
array([[0.7529412 , 0.6392157 , 0.6745098 , …, 0.6901961 , 0.65882355,
        0.7019608 ],
       [0.75686276, 0.6431373 , 0.6745098 , …, 0.6745098 , 0.64705884,
        0.69411767],
       [0.7607843 , 0.64705884, 0.6745098 , …, 0.69803923, 0.6862745 ,
        0.7254902 ],
       …,
       [0.6       , 0.5882353 , 0.64705884, …, 0.7764706 , 0.8       ,
        0.7529412 ],
       [0.6117647 , 0.59607846, 0.654902  , …, 0.78039217, 0.80784315,
        0.7607843 ],
       [0.67058825, 0.64705884, 0.69803923, …, 0.78431374, 0.8117647 ,
        0.7647059 ]], dtype=float32)
```

Il s'agit d'un tableau numpy 2d, avec des nombres allant de 0 à 1 : 0 pour un pixel noir, 1 pour un pixel blanc, et entre ces valeurs toutes les nuances de gris. En général, il existe bien plus que 50 nuances : les images sont encodées en 8 bits, c'est-à-dire que chaque pixel peut prendre une valeur parmi $2^8 = 256$ valeurs différentes (de 0 à 255). Pour une image en couleurs, on définit ces 256 valeurs pour chacune des couleurs de la synthèse

primaire (Rouge, Vert et Bleu) : les images couleurs contiennent donc trois fois plus d'informations.

2.3.2 Transformations géométriques

Dans la suite l'énoncé propose différentes transformations, classées par ordre croissant de difficulté. En choisir au moins une à faire, puis la tester sur l'image fournie, ou tout autre image de votre choix.

Rotation : Ecrire un code permettant de créer puis afficher une nouvelle image, qui est simplement l'image de départ, mais qui a subi une rotation de 90° vers la gauche.

[7]:
```
Image_90 = np.zeros((211,268))
for i in range(211):
    for j in range(268) :
        Image_90[i,j] = Image[j,i]
ratio, dpi =  1.5, 70
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.gray()
plt.imshow(Image_90)
plt.show()
```

Symétrie : Ecrire un code permettant de créer puis afficher une nouvelle image, qui est simplement l'image de départ, mais qui a subi une symétrie gauche-droite (symétrie d'axe l'axe vertical coupant l'image en 2).

[9]:
```
Image_sym_v = np.zeros((268,211))
for i in range(268):
    for j in range(211) :
        Image_sym_v[i,j] = Image[i,210-j]
ratio, dpi =  1.5, 70
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.gray()
plt.imshow(Image_sym_v)
plt.show()
```

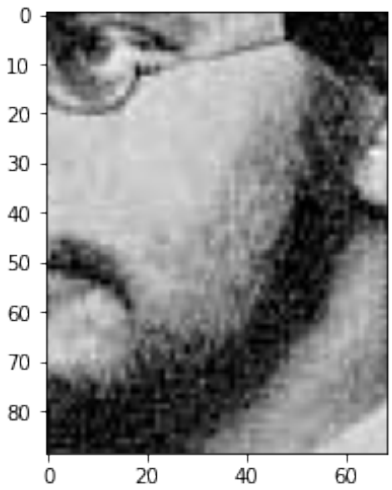
Rognage : Ecrire un code permettant de créer puis afficher une nouvelle image rognant l'image de façon centrée, en gardant son ratio inital, selon le poucentage de l'image initial donné en argument. Plus explicitement, l'image finale doit présenter le centre de l'image initial, de façon à ce que ses dimensions horizontales et verticales soit le pourcentage (donné en argument) des dimensions initiales.

[5]:
```
pour_cent = 33

# Nouvelles dimensions de l'image :
lignes = 268 * pour_cent // 100
colonnes = 211 * pour_cent // 100

milieu = (268//2, 211//2)

Image_rog = Image[milieu[0] - lignes//2 : milieu[0] + lignes//2 + 1, milieu[1] -␣
 ↪colonnes//2 : milieu[1] + colonnes//2 + 1]
ratio, dpi =  1.5, 70
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.gray()
plt.imshow(Image_rog)
plt.show()
```



2.3.3 Traitement de l'image

Dans la suite l'énoncé propose différentes modifications de l'image, classées par ordre croissant de difficulté ? En choisir au moins une à faire, puis la tester sur l'image fournie, ou tout autre image de votre choix.

Noir et Blanc : Ecrire un code permettant de créer puis afficher une nouvelle image, qui transforme l'image initial en une image en noir et blanc. On pourra créer une fonction ayant pour argument un seuil, compris entre 0 et 1, permettant de discerner, par comparaison à ce seuil, les pixels blancs des pixels noirs.

[23]:
```
seuil = 0.6
Image_NB = np.zeros((268,211))
for i in range(268):
    for j in range(211) :
        if Image[i,j] > seuil :
            Image_NB[i,j] = 1
        else :
            Image_NB[i,j] = 0
ratio, dpi =  1.5, 70
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.gray()
plt.imshow(Image_NB)
plt.show()
```

Postérisation : Ecrire un code permettant de créer puis afficher une nouvelle image, qui transforme l'image initial en une image contenant toujours des niveaux de gris, mais moins que 256. On pourra créer une fonction ayant pour argument le nombre de niveau de gris désiré.

[12]:
```
def posterisation(n) :
    Image_post = np.zeros((268,211))
    for i in range(268):
        for j in range(211) :
            Image_post[i,j] = int(Image[i,j]*n)/n
    ratio, dpi =  1.5, 70
    plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
    plt.gray()
    plt.imshow(Image_post)
    plt.show()
```

[25]:
```
posterisation(4)
```

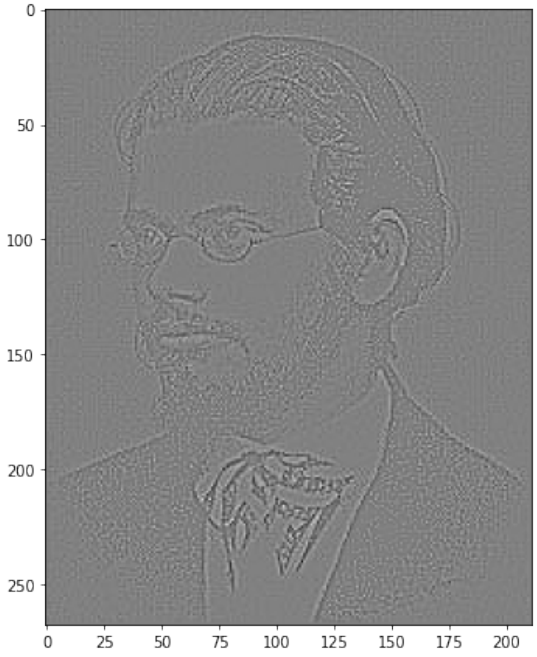
Floutage : Ecrire un code permettant de créer puis afficher une nouvelle image, qui transforme l'image initial en une image floutée : celle-ci remplace la valeur de la luminosité en chaque pixel (i, j) par la moyenne de la luminosité des 9 pixels voisins (situés entre $i-1$ et $i+1$, et entre $j-1$ et $j+1$). On pourra utiliser la fonction `np.sum(Tab)` permettant de sommer les éléments de `Tab`.

[10]:
```
Image_floue = np.zeros((268,211))
for i in range(268):
    for j in range(211) :
        Image_floue[i,j] = np.sum(Image[i-1:i+2,j-1:j+2])/9
ratio, dpi =  1.5, 70
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.gray()
plt.imshow(Image_floue)
plt.show()
```

Mais au fait, que se passe-t-il au niveau des bords ? -> périodicité....

Traitement mystère : Ecrire un code permettant de créer puis afficher une nouvelle image, qui transforme l'image initial en remplaçant la valeur de la luminosité de chaque pixel (i, j) par 4 fois la luminosité de ce même pixel, à quoi on retranche la luminosité des 4 plus proches voisins. Une tablette de chocolat à celui ou celle qui parvient à comprendre cette modification !

[21]:
```
Image_mystere = np.zeros((268,211))
for i in range(1,267):
    for j in range(1,210) :
        Image_mystere[i,j] =␣
 ↪4*Image[i,j]-Image[i-1,j]-Image[i+1,j]-Image[i,j-1]-Image[i,j+1]
ratio, dpi =  1.5, 70
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.gray()
plt.imshow(Image_mystere)
plt.show()
```


0
50
100
150
200
250
0
25
50
75
100
125
150
175
200