MPRA
Munich Personal RePEc Archive

Data Science with Python: Algorithm, Statistics, DataViz, DataMining and Machine-Learning

Keita, Moussa

February 2017

Online at https://mpra.ub.uni-muenchen.de/76653/
MPRA Paper No. 76653, posted 07 Feb 2017 14:56 UTC

Data Science sous Python

Algorithme, Statistique, DataViz, DataMining et Machine-Learning

Par

Moussa Keita, PhD

Consultant Big Data-Data Science

Umanis Consulting Group, Paris

Février 2017

(Version 1.0)

Résumé

La Data Science est une discipline technique qui associe les concepts statistiques aux algorithmes puissants de calculs informatiques en vue du traitement et de la modélisation des données de masse issues des phénomènes de l'observation (économiques, industriels, commerciaux, financières, managériaux, sociaux, etc..). En matière de Business Intelligence et de veille stratégique, la Data Science est devenue un outil incontournable d'aide à la prise de décisions pour les dirigeants. Elle permet d'exploiter et de valoriser tout le patrimoine informationnel interne et externe de l'entreprise. Le langage de programmation Python s'est rapidement imposé, au cours des récentes années, comme un langage de premier choix à la disposition des Data Scientists pour exploiter le potentiel grandissant du Big Data. Le gain de popularité dont bénéficie, aujourd'hui, ce langage s'explique en grande partie par les nombreuses possibilités offertes par ses puissants modules notamment d'analyses numériques et de calculs scientifiques (numpy, scipy, pandas), de visualisation de données (matplotlib) mais aussi de Machine Learning (scikit-learn). Présenté dans une approche hautement pédagogique, ce manuscrit revisite les différents concepts indispensables à la maitrise de la Data Science sous Python. Le travail est organisé en sept chapitres. Le premier chapitre est consacré à la présentation des bases de la programmation sous Python. Dans ce chapitre, nous présentons notamment les différents éléments de langage qui forment l'ossature du programme python mais aussi les différents objets de programmation qui en forment le socle. Le second chapitre est consacré à l'étude des chaînes de caractères et des expressions régulières. L'objectif de ce second chapitre est de se familiariser avec le traitement et l'exploitation des chaînes de caractères qui constituenet les valeurs des variables couramment rencontrées dans les bases de données non structurées. Le troisième chapitre est consacré à la présentation des méthodes de gestion de fichiers et de traitements de texte. Le but de ce chapitre est d'approfondir le précédent chapitre en présentant les méthodes couramment utilisées pour le traitement des données non structurées qui se présentent généralement sous forme de fichiers de textes. Le quatrième chapitre est consacré à la présentation des méthodes de traitement et d'organisation des données initialement stockées sous forme de tables de données. Le cinquième chapitre est consacré à la présentation des méthodes classiques d'analyses statistiques (analyses descriptives, tests statistiques, régression linéaires et logistiques,...). Quant au sixième chapitre, il est consacré à la présentation des méthodes de datavisualization : histogrammes, diagrammes en barres, pie-plots, box-plots, scatter-plots, courbes d'évolution,

graphiques en 3D,...). Enfin le septième chapitre est consacré à la présentation des méthodes de datamining et de machine-learning. Dans ce chapitre, nous présentons notamment les méthodes de réductions de dimensions des données (Analyses en Composantes Principales, Analyses Factorielles, Analyses des Correspondances Multiples) mais aussi des méthodes de classification (Classification Ascendante Hiérarchique, K-Means Clustering, Support Vector Machine, Random Forest). Toutefois, nous tenons à signaler que le manuscrit est toujours en progression. De ce fait nous restons ouverts à toutes les critiques et suggestions de nature à améliorer son contenu[*].

[*] **Contact info**: Email : keitam09@ymail.com

Nous tenons ici à remercier les auteurs dont les travaux ont été d'un apport considérable à la rédaction de ce document. Il s'agit notamment de Cordeau et Pointal (2010), Fuchs et Pierre (2014), Le GOFF(2011), Rakotomalala Ricco, (2015) , Swinnen Gérard,(2010) ainsi que Chris Albon à travers son site http://chrisalbon.com/.

Codes JEL: C8

Mots clés: Programmation, langage Python, Data science, Traitement et analyses de données, data visualization.

Table des matières

Chapitre 1 : Les bases de la programmation sous python 21

1.1. Introduction 21

1.2. Les éléments structurants d'un programme Python 22

1.2.1. Les variables 22

1.2.1.1. Définir une variable 22

1.2.1.2. Type d'une variable 25

1.2.1.3. Méthodes associées aux variables 27

1.2.1.3. Les opérateurs arithmétiques et logiques (booléens) 28

1.2.2. Les instructions conditionnelles : la clause « if… else » 30

1.2.3. Les instructions en boucle 32

1.2.3.1. Les instructions en boucle « while… ».. 32

1.2.3.2. Les instructions en boucle « for… in… ».. 33

1.2.4. Les instructions break et continue dans une clause « while… » ou « for … in…» 35

1.2.5. Les mots réservés du langage python 35

1.3. Les principaux objets du langage Python 36

1.4. Etude des objets « liste » 38

1.4.1. Caractéristiques générales des listes 38

1.4.1.1. Définition d'un objet liste 39

1.4.1.2. Indiçage d'une liste (slicing) 40

1.4.3. Les opérations sur les listes 41

1.4.3.1. Déterminer le nombre d'éléments d'une liste 41

1.4.3.2. Additionner des listes 42

1.4.3.3. Démultiplication des éléments d'une liste 42

1.4.3.4. Modifier ou remplacer les éléments d'une liste 43

1.4.3.5. Ajouter de nouveaux éléments à une liste : fonction append() 43

1.4.3.6. Insérer de nouveaux éléments dans une liste : fonction insert() 44
1.4.3.7. Supprimer un élément d'une liste fonctions remove() ou delete 44
1.4.3.8. Renvoyer l'indice d'un élément en se basant sur sa valeur :fonction index() 45
1.4.3.9. Compter le nombre d'occurrences d'une valeur dans une liste : fonction count()........... 45
1.4.3.10. Trier les éléments d'une liste : fonction sort()... 45
1.4.3.11. Intervertir l'ordre des éléments dans une liste : fonction reverse()................................ 46
1.4.3.12. Joindre les éléments d'une liste pour former une chaîne de caractères : fonction join()46
1.4.3.13. Tester l'existence d'un élément dans une liste : fonction in 47
1.4.3.14. Récupérer simultanément les indices et les valeurs des éléments d'une liste : fonction enumerate() 47
1.4.3.15. Faire la copie d'une liste 48
1.4.3.16. Combiner les éléments de deux listes pour former une liste de couple d'éléments : fonction zip() 51

1.5. Les fonctions map(), filter() et reduce() pour le traitement des listes 53

1.5.1. La fonction map() 53

1.5.2. La fonction filter() 55

1.5.3. La fonction reduce()... 56

1.6. Etude des objets « tuples » 57

1.6.1. Définition d'un tuple 58

1.6.2. Indiçage des tuples (slicing)... 58

1.7. Etude des objets « set » 59

1.8. Etude des objets « array » 60

1.8.1. Définition d'un array 61

1.8.1.1. Définition d'un array à 1 dimension 61
1.8.1.2. Définition d'un array à plusieurs dimensions 62
1.8.1.3. Indiçage d'un array (slicing) 62

1.8.2. Déterminer la dimension d'array 63

1.8.3. Les opérations sur les arrays 63

1.8.3.1. Conversion d'un array vectoriel en array matriciel: fonction reshape() et resize()........... 63

1.8.3.2. Transposé d'un array matriciel: fonction transpose() 64
1.8.3.3. Création d'un array matriciel rempli de 0 ou 1: fonctions zeros() et ones() 64
1.8.3.4. Les opérations algébriques (matricielles) sur les arrays 64
1.8.3.5. Utilisation de la fonction argsort() sur un array 65
1.9. Etude des objets « dictionnaire » 65
1.9.1. Définition d'un dictionnaire 65
1.9.2. Accéder aux éléments d'un dictionnaire (slicing) 66
1.9.2.1. Slicing d'un dictionnaire dans le cas d'une clé à valeur unique 66
1.9.2.2. Slicing d'un dictionnaire dans le cas d'une clé à plusieurs valeurs 67
1.9.3. Modification de dictionnaire : ajout et suppression de clés ou de valeurs 67
1.9.3.1. Ajout ou modification de clés ou de valeurs 67
1.9.3.2. Suppression de clés ou de valeurs dans un dictionnaire 68
1.9.4. Renommer une clé dans un dictionnaire 68
1.9.5. Tester l'existence d'une clé dans un dictionnaire : la fonction in 68
1.9.6. Récupérer les clés ou les valeurs d'un dictionnaire (les fonctions keys() et values()) 69
1.10. Etude des fonctions 69
1.10.1. Aperçu général sur quelques fonctions prédéfinies sous python : la fonction print() et la fonction input() 70
1.10.1.1. La fonction print() 70
1.10.1.2. La fonction input() 70
1.10.2. Les fonctions définies par l'utilisateur 71
1.10.2.1. Définir une fonction 71
1.10.2.2. Les fonctions lambda 77
1.10.3. Les variables locales et les variables globales 77
1.10.4. Récupérer la valeur renvoyée par une fonction : l'instruction return ... 79
1.10.5. Utilisation des générateurs de fonction : l'instruction yield 80
1.10.6. Gestion des erreurs et des exceptions lors de la définition d'une fonction 82

1.10.6.1. Utilisation des blocs ty-except 82
1.10.6.2. Utilisation des clause « else » et « finally » dans une exception 85
1.10.6.3. Les exceptions déclenchées par l'utilisateur : l'instruction raise 86
1.10.7. Documenter une fonction 87
1.11. Les modules python 87
1.11.1. Importation d'un module python 88
1.11.2. Utilisation d'un module python 89
1.11.2.1. Quelques utilisations de la fonction math 89
1.11.2.2. Quelques exemples d'utilisation du module random 89
1.11.3. Définir et utiliser son propre module 91
1.11.4. Procédure d'installation et d'utilisation des modules externes 91
1.11.5. Désinstaller un module 93
1.12. Aperçu général sur les objets « classe » 94
1.13. Utilisation du module pickle pour l'enregistrement et le chargement des objets python 95
1.13.1. Utilisation de la fonction Pickler 95
1.13.2. Utilisation de la fonction Unpickler 96
Chapitre 2 : Etude des chaînes de caractères et des expressions régulières 97
2.1. Les chaînes de caractères 97
2.1.1. Définition d'une variable de type chaîne de caractères 97
2.1.2. Indiçage des chaînes de caractères (slicing) 98
2.1.3. Déterminer la longueur d'une chaîne de caractères (nombre de caractères) 99
2.1.4. Addition de chaînes de caractères (concaténation) 99
2.1.5. Modifier la casse d'une chaîne en majuscule, minuscule ou capital : fonctions upper(), lower() et capitalize() 100
2.1.6. Rechercher un caractère (ou un motif) dans une chaîne et renvoyer son indice : fonction find() 100

2.1.7. Rechercher un caractère (un motif) dans une chaîne et remplacer par un autre : fonction replace() 102

2.1.8. Compter le nombre d'occurrence d'un motif dans une chaîne : fonction count() 102

2.1.9. Découper une chaîne de caractères en liste : fonction list() ou fonction split() 102

2.1.9.1. Cas où les espaces sont traités comme des valeurs (list()) 102

2.1.9.2. Cas où les espaces sont traités comme des séparateurs (split()) 103

2.1.10. Les opérations logiques et booléennes sur les chaines de caractères. 104

2.1.10.1. La fonction startwith() 104

2.1.10.2. la fonction endwith() 105

2.1.10.3. La fonction in 105

2.1.10.4. La fonction islower() 105

2.1.10.5. La fonction isupper() 105

2.1.10.6. La fonction istitle() 105

2.1.10.7. La fonction isalpha() 106

2.1.10.8. La fonction isalnum() 106

2.1.10.9. La fonction isdigit() 106

2.1.11. Formatage de valeurs à l'intérieur d'une chaine de caractères 107

2.1.11.1. Formatage avec l'opérateur de concaténation "+" 108

2.1.11.2. Utilisation de l'opérateur de formatage "%" % 108

2.1.11.3. Utilisation de la fonction format() 109

2.1.11.4. Formatage de valeurs indicées et création de variables indicées 110

2.1.12. Utilisation de l'opérateur antislash \ : mise en forme de la chaîne et traitement des caractères spéciaux 111

2.2. Etudes des expressions régulières (regex ou re) 112

2.2.1. Définition et rôle des expressions régulières 112

2.2.2. Les opérateurs d'expression régulière 113

2.2.3. Exemples d'application des opérateurs d'expressions régulières 115

2.2.4. Construire une expression : la fonction compile() 117

2.2.4.1. Le rôle de la fonction compile() 117

2.2.4.2. Utilisation de la fonction compile() sans directives 117
2.2.4.3. Utilisation de la fonction compile() avec directives 118
2.2.5. Quelques fonctions associées aux expressions régulières 118
2.2.5.1. La fonction search() 118
2.2.5.2. La fonction match() 120
2.2.5.3. La fonction findall() 121
2.2.5.4. La fonction fonction sub() 123
2.2.5.5. La fonction findinter() 123
2.2.5.6. La fonction group() 124
2.2.6. Etude de quelques expressions régulières couramment rencontrées .. 126
2.2.6.1. Cas 1 : Correspondance avec une expression exacte uniquement 126
2.2.6.2. Cas 2 :Correspondance avec un mot ou une expression figurant dans une liste 126
2.2.6.3. Cas 3 :Correspondance avec un mot présentant des variantes orthographiques ou des caractères spéciaux 127
2.2.6.4. Cas 4 : Correspondance avec une adresse e-mail sur un domaine spécifique 127
2.2.6.5. Cas 5 :Correspondance avec une adresse IP comprise dans une plage 128
2.2.6.6. Cas 6 : Correspondance avec un format alphanumérique 129
2.2.7. Exercices de synthèse sur l'utilisation des regex 130
Chapitre 3 : Gestion de fichiers et traitements de texte sous Python........133
3.1. Aperçu général sur quelques modules de gestion de fichiers sous Python 133
3.2. Utilisation du module o.s pour la gestion des fichiers et des répertoires........ 133
3.2.1. Définir un répertoire de travail........ 133
3.2.2. Créer un nouveau dossier dans le répertoire défini 134
3.2.3. Renommer un dossier dans le répertoire 134
3.2.4. Supprimer un dossier........ 134
3.2.5. Tester si un fichier existe dans le répertoire 134
3.2.6. Supprimer un fichier exitant dans le répertoire 135

3.2.7. Lister l'ensemble des fichiers présents dans un répertoire (avec ou sans extensions) 136
3.2.8. Découper le chemin complet : la fonction os.path.split() 136
3.2.9. Recomposer le chemin complet à partir d'un chemin et du nom du fichier : la fonction os.path.joint() 136
3.2.10. Tester si un chemin existe 137
3.2.11. Tester si un chemin conduit à un fichier 137
3.2.12. Tester si le chemin indiqué est un répertoire 137
3.2.13. Ouvrir un fichier 137
3.3. Gestion de répertoires et de fichiers : utilisation du module shutil 139
3.3.1. Copie de dossier ou de fichiers : avec le module shutil 139
3.3.1.1. Utilisation de la fonction shutil.copyfile() 139
3.3.1.2. Utilisation de la fonction shutil.copy 139
3.3.1.3. Utilisation de la fonction shutil.copyfileobj 139
3.3.1.4. Utilisation de la fonction shutil.copystat 140
3.3.2. Création de fichier archive ZIP ou TAR avec shutil 140
3.4. Visualiser le contenu du fichier lu : les fonctions read(), read(n), readline(), readlines(), et xreadlines() 141
3.4.1. Utilisation de la fonction read() 141
3.4.2. Utilisation de la fonction readline() 141
3.4.3. Utilisation de la fonction readlines() 142
3.4.4. Utilisation de la fonction read(n) 142
3.4.5. Utilisation de la fonction xreadlines() 142
3.4.6. Gestion des lignes en blank générées par les commandes readline() et readlines() 142
3.5. Découper un texte en une liste de mots : la fonction splitlines() ou la fonction append() 143
3.5.1. Utilisation de la fonction splitlines() 143

3.5.2. Utilisation de la boucle for avec la fonction append() 143
3.6. Déplacer le curseur dans un texte : les fonctions seek() et tell() 145
3.7. Ouvrir et modifier le contenu d'un fichier : la fonction write() et writelines() 145
3.8. Ouvrir un fichier en mode append 'a' 148
3.9 Ouvrir un fichier en mode binaire 'b' 149
3.10. Utilisation de la fonction « with ... as » pour la gestion des fichiers 149
3.11. Encodage et décodage des textes en python 149
3.11.1. Décodage de texte 150
3.11.2. Encodage de texte 151
3.11.3. Peut-on détecter l'encodage d'un fichier texte ? 152
3.12. Présentation du module URLLIB pour gestion des urls et la lecture de fichiers à partir des urls (uniform resource locator) 153
3.12.1. Le module urllib.request 153
3.12.1.1. Utilisation de la fonction urllib.request.urlopen() 153
3.12.1.2. Utilisation de la fonction urllib.request.urlretrieve() 154
3.12.1.3. Utilisation de la fonction urllib.request.Request() 154
3.12.1.4. Cas des requêtes POST 155
3.12.2. Utilisation du module requests comme alternatif de urllib.request 155
3.12.3. Le module urllib.error 156
3.12.3.1. Utilisation de la fonction urllib.error.URLError() 156
3.12.3.2. Utilisation de la fonction HTTPError() 156
3.12.4. Le module urllib.parse() 156
3.12.4.1. Décomposer un url en différentes composantes 156
3.12.4.2. Rassembler différentes composantes en un url 157
3.13. Extraction et gestion des textes a partir des pages html ou xml (pages avec balisages) : utilisation du module BeautifulSoup 158

3.13.1. Utilisation du module BeautifulSoup: extraction du texte à partir d'une page htlm ou xlm avec parsing 158

3.13.1.1. Lecture de la page html ou xml 158

3.13.1.2. Le parsing de la page récupérée 158

3.13.1.3. Navigation dans le texte parsé 160

3.13.2. Quelques balises html (tags) 162

3.13.3. Utilisation du module pattern-web pour le parsing des pages web.... 164

3.13.4. Utilisation du module xml pour le parsing des pages web 164

3.13.5. Utilisation du module html2text pour le parsing des pages web 164

3.13.6. Utilisation du module NLTK pour le parsing des pages web 164

3.14. Text mining avec le module NLTK 165

3.14.1. Convertir un texte en format nltk 165

3.14.2. Découper une chaîne de caractères en des mots : la fonction tokenize() 166

3.14.3. Rechercher un mot dans un texte et afficher la (les) phrase(s) qui le contient(nent) 167

3.14.4. Repertorier les n premiers mots les plus fréquents dans un texte avec la fonction FreqDist 170

3.14.5. Compter le nombre d'occurences de la longueur des mots dans un texte 170

3.14.6. Identifier des collocations de mots 171

3.14.7. Stemming, lemmatisation et post-tag des mots dans un texte 171

3.14.7.1. Le stemming ou la racinisation 171

3.14.7.2. La lemmatisation 173

3.14.7.3. Les étiquettes grammaticales (les post-tags) 173

3.14.7.4. Traitement des StopWords (les mots vides) 174

3.15. Text Matching : calcul des Edit distances et des ratios de matching entre deux séquences de caractères 174

3 .15.1. Le module difflib 174

3 .15.1.1. Utilisation de la fonction SequenceMatcher() 175

3 .15.1.2. Utilisation des fonctions get_matching_blocks() et get_opcodes() après SequenceMatcher() 176
3 .15.1.3. Utilisation de la fonction differ() 176
3 .15.1.4. Utilisation de la fonction ndiff() 177
3 .15.1.5. Utilisation de la fonction unified_diff() 177
3 .15.1.6. Utilisation de la fonction context_diff() 177
3 .15.1.7. Utilisation de la fonction Htmldiff() 177
3 .15.2. Le module Levenshtein 178
3.15.3. Le module distance 179
3.15.3.1. Distance de Levenshtein 179
3.15.3.2. Distance de Hamming 179
3.15.3.3.Distance de Sorensen 179
3.15.3.4.Distance de jaccard 180
3.15.4. Le module FuzzyWuzzy 180
3.15.5. Comparaison des différentes méthodes de matching 181
3.16. Text Clustering : regroupement des termes d'un texte par des algorithmes de clustering 182
3.16.1. Text clustering avec l'algorithme de l'Affinity Propagation 182
3.16.2. Text clustering par des algorithmes basés sur la matrice TF-IDF 185
3.16.2.1. Construction de la matrice TF-IDF 185
3.16.2.2. Application du K-means clustering à partir de la matrice de distance 190
3.16.2.3. Application de la Classification Ascendente Hierarchique (CAH) à partir de la matrice de distance 192
3.16.2.4. Application du Multidimensional scaling (MDS) à partir de la matrice de distance 193
3.17. Stratégie générale de pré-traitement de texte en vue de la recherche d'information, du text matching ou du text clustering . 195
Chapitre 4 : Traitement et Organisation des tables de données 197
4.1. Création des objets « series » 197
4.1.1. Création d'une série à partir d'une liste de valeurs 197

4.1.2. Création d'une série à partir d'un dictionnaire 197
4.1.3. Définir des indices pour les séries (identifiant des séries) 198
4.2. Création de table de données (data frame) 198
4.2.1. Création de data frame à partir des séries 199
4.2.2. Création du dataframe à partir d'un dictionnaire 199
4.2.3. Création d'un dataframe vide (d'observations) 200
4.3. Création de dataframe à partir des sources de données externes 200
4.3.1. Importation des fichiers txt et csv 200
4.3.2. Importation de fichiers Excels 203
4.3.3. Importation de données à partir des sources de données non structurées 203
4.3.3.1. Exemple 1 : Convertir un ensemble de texte en un dataframe avec des méthodes classiques de traitement de texte 204
4.3.3.2. Exemple 2 : Convertir un ensemble de texte en dataframe en utilisant les expressions régulières (regex) 205
4.4. Exportation des données vers des formats externes 206
4.4.1. Exportation vers le format csv : 206
4.4.2. Exportation vers le format Excel (utilisation de l'engine xlswriter 206
4.5. Description de la table de données 208
4.6. Paramétrer le nombre de colonnes et le nombre de lignes à afficher lors de l'affichage de la table de données 209
4.7. Opérations sur les variables 209
4.7.1. Sélectionner les variables 209
4.7.1.1. Sélectionner une seule colonne 209
4.7.1.2. Sélectionner plusieurs colonnes 209
4.7.2. Renommer une colonne dans un dataframe 210
4.7.3. Transformer les noms des variables en miniscule ou en majuscule 210
4.7.4. Transformer une colonne en un index dans un dataframe 210

4.7.5. Extraire les noms des colonnes sous forme de liste 210

4.7.6. Types des variables dans un dataframe 211

4.7.6.1. Déterminer le type des variables : la fonction dtypes 211

4.7.6.2. Convertir le type d'une variable dans un dataframe : la fonction astype() 211

4.7.7. Création de variable 212

4.7.7.1. Création des variables avec les opérateurs simples 212

4.7.7.2. Création des variables avec les fonctions mathématiques 213

4.7.7.3. Création des variables par recodage : recodage conditionnel 214

4.7.7.4. Recodage d'une variable continue (par intervalle de valeurs) 217

4.7.8. Discrétisation d'une variable quantitative 217

4.7.9. Convertir une variable catégorielle en une variable binaire 218

4.7.9.1. Utilisation de la fonction get_dummies() 218

4.7.9.2. Utilisation de la fonction patsy.dmatrix() 218

4.7.9.3. Utilisation de la fonction patsy.dmatrix() pour une variable numérique 219

4.7.10. Renommer les modalités d'une variable catégorielles 219

4.7.11. Suppression de variables 219

4.7.11.1. Supprimer une seule variable 219

4.7.11.2. Suppression de plusieurs variables 219

4.7.11.3. Supprimer toutes les variables dont le nom commence par un mot 220

4.7.12. Récupérer les valeurs d'une variable et les stocker sous forme de liste 220

4.7.13. Convertir une date en chaine de caractères en une date reconnue 221

4.8. Opérations sur les observations 221

4.8.1. Sélectionner des observations 221

4.8.1.1. Sélection des observations à partir de leur indice 221

4.8.1.2. Sélction des observations selons les valeurs d'une ou de plusieurs variables (sélection conditionnelle) 221

4.8.1.3. Sélectionner des observations par tirage aléatoire 222

4.8.2. Trier les observations 223

4.8.3. Traitement des observations avec valeurs manquantes 224

4.8.3.1. Création d'une table avec des valeurs manquantes 224
4.8.3.2. Supprimer toutes les observations avec valeurs manquantes 224
4.8.4. Identifier l'observation qui a la valeur maximale sur une variable 225
4.8.5. Suppression d'observations 225
4.8.5.1. Suppression des observations sur la base des indices 226
4.8.5.2. Suppression des observations selon une condition 226
4.8.6. Identification et Suppressions des obervations dupliquées dans une table 226
4.8.6.1. Identification des observations dupliquées 226
4.8.6.2. Suppression des observations dupliquées 226
4.8.7. Comptage des valeurs dans une table 227
4.8.7.1. Comptage des valeurs sur une seule variable 227
4.8.7.2. Comptage sur plusieurs variables 227
4.8.8. Créer un rang pour les obervations 227
4.9. Construire une boucle DO LOOP (sur les variables ou sur les observations) 228
4.10. Calculer la somme par colonne ou par ligne et calcul du cumul 228
4.10.1. Calcul de la somme par ligne 229
4.10.2. Calcul de la somme par colonne 229
4.10.3. Calcul de la somme cumulée par ligne ou par colonne 229
4.11. Calcul de valeurs par groupe et agrégation des données dans une table 229
4.11.1. Utilisation de la fonction groupby() 229
4.11.2. Utilisation de la fonction groupby() combinée avec la fonction agg() 230
4.11.3 Utilisation d'une formule dans la définition d'une fonction d'aggréation 231
4.11.4. Définition d'une agréagtion plus complexe: statistique pour pour plusieurs variable (avec plusieurs fonctions par variable) 231
4.11.5. Merging des valeurs agrégées à la table initiale 232

4.12. Opération sur tables de données 233

4.12.1. Fusion de tables 233

4.12.1.1. Fusion verticale ou juxtaposition de table (append) 234

4.12.1.2. Fusion de deux tables sur les variables (sans clé d'identification) 234

4.12.1.3. Fusion de deux tables sur les variables avec clé d'identification (merge) 234

4.12.2. Reformattage de tables (reshape) 236

4.12.3. Mettre une table sous format verticale 236

4.13. Standardiser les variables d'une table 237

Chapitre 5 : Les analyses statistiques classiques sous Python 239

5.1. Paramétrage de python et chargement des modules usuels .. 239

5.2. Statistiques descriptives 239

5.2.1. Statistiques desriptives sur les variables quantitatives 239

5.2.1.1. Utilisation du scipy.stats 239

5.2.1.2. Utilisation du module statsmodels 241

5.2.2. Statistiques descriptives sur les variables qualitatives 241

5.2.2.1. Tableau de fréquence univarié (tri simple) 241

5.2.2.2. Tableau de fréquences croisé 242

5.2.3. Statistiques descriptives croisées et mesures d'associations 243

5.2.3.1. Mesure d'assocation entre variables quantitatives : le coefficient de corrélation 243

5.2.3.2. Mesure d'association entre variables qualitatives : le test d'indépendance de khi-deux 244

5.3. Calcul de quantiles, de fonction de répartition et de fonction de densité 244

5.3.1. Fonction de répartition et calcul de quantile 244

5.3.2. Lecture des tables statistiques avec python 245

5.3.2.1. Loi normale centrée et réduite 245

5.3.2.2. Loi de Student 245

5.3.2.3. Loi du khi-2 246

5.3.2.4. Loi de Fisher 246
5.4. Tests d'adéquation à une Loi 246
5.4.1. Test d'adéquation à une loi normale 246
5.4.1.1. Test de normalité d'Agostino 246
5.4.1.2. Test de Normalité Shapiro-Wilks 246
5.4.1.3. Test de normalité d'Anderson-Darling 247
5.4.1.4. Test de normalité de Kolmogorov-Smirnov 247
5.4.1.5. Le test de normalité de Lillifors 247
5.4.2. Test d'adéquation à une loi de khi-deux 247
5.5. Test de conformité à valeur de référence : test d'égalité de la moyenne à une valeur 248
5.6. Test d'égalité de moyennes sur échantillons indépendants 249
5.7. Test d'égalité de moyennes sur échantillons indépendants appariés 249
5.8. Comparaison de moyennes sur plusieurs groupes : le test ANOVA 250
5.9. Modèles de régressions linéaires multiples 250
5.9.1. Estimation d'un modèle de régression linéaire 250
5.9.1.1. Utilisation du module statsmodels 250
5.9.1.2. Utilisation du module sklearn 252
5.9.2. Analyse prédictive à partir du modele de régression linéaire avec le module sklearn 253
5.9.2.1. Estimation et validation du modèle 253
5.10. Régression logistique binaire 255
5.10.1. Estimation du modèle logistique binaire 255
5.10.1.1. Utilisation du module statsmodels 255
5.10.1.2. Utilisation du module sklearn 256
5.10.2. Selection automatique des variables explicatives dans la regresssion logistique binaire 257

5.10.3. Analyse prédictive à partir du modèle logistique binaire avec le module sklearn 259

5.10.4. Segmentation, scoring et ciblage avec le modele logistique binaire.. 261

5.10.4.1. Objectifs généraux 261

5.10.4.2. Mise en œuvre de la démarche du ciblage 261

5.11. Estimation de modèle de régression logistique multinomiales 263

Chapitre 6 : Data visualisation avec python 264

6.1. Les graphiques à barres 264

6.1.1. Barres simples de comparaison de moyennes 264

6.1.2. Paramétrage du graphique 267

6.1.2.1. Description du role de fig, ax= dans un tracé de graphique 267

6.1.2.2. Liste des couleurs en python 268

6.1.2.3. Ajouter les valeurs des barres au graphique 268

6.1.2.4. Définition et paramétrage des graphiques en plusieurs cadrans 269

6.1.3. Barres de comparaison (de valeurs) par groupe 270

6.1.4. Barres de comparaison (de moyennes) par groupe 271

6.1.5. Barres inversées (modèle pyramides des âges) 275

6.1.6. Superposition de plusieurs graphiques en barres dans une fenêtre graphique unique 277

6.1.6.1. Tracer des deux graphiques en barres dans le même cadran 278

6.1.6. 2. Tracer des deux graphiques en barres dans deux cadrans séparés 279

6.2. Les Histogrammes 280

6.2.1. Histogrammes simples et histogrammes combinés 280

6.2.2. Combinaison de plusieurs histogrammes dans un seul cadre graphique 283

6.3. Les diagrammes circulaires 291

6.3.1. Diagramme circulaire simple 291

6.3.2. Regrouper plusieurs diagrammes circulaires dans un seul graphique .. 293

6.4. Les diagrammes de fréquences 294
6.4.1. Diagramme de fréquence sur une variable qualitative à codage numérique (ou sur une variable quantitative)... 295
6.4.2. Diagramme de fréquence sur une variable qualitative à codage en caractères...296
6.5. Les graphiques en nuages de points 297
6.5.1. Graphique simple de nuage de points 297
6.5.2. Labéliser les valeurs dans un nuage de points..................................299
6.5.3. Ajouter une ligne verticale ou horizontale à un nuage de point 300
6.5.4. Nuage de points selon des catégories (représentation dans le même cadran)... 301
6.5.5. Nuage de points selon des catégories (représentation dans des cadrans différents) 303
6.6. Graphique en courbe d'évolution (ligne) 306
6.6.1. Courbe d'évolution d'une seule variable..306
6.6.2. Représenter plusieurs courbes dans un même cadran 307
6.6.3. Représenter plusieurs courbes (dans des cadrans différents) 308
6.7. Graphiques en box-plot 309
6.7.1. Box plot simple sur une seule variable..309
6.7.2. Box plots par catégorie ou pour plusieurs variables (representés dans le même cadran).. 311
6.7.3. Box plots par catégorie ou pour plusieurs variables (représentés dans des cadrans différents) 313
6.8. Les graphiques en 3D: exemple: Nuage de points en 3D 316
Chapitre 7: Data mining et machine-learning sous python 318
7.1. Analyses en composantes principales (ACP) 318
7.2. Analyses factorielles (AFC).. 319

7.3. Analyses des correspondances multiples (ACM) 320
7.4. Classification ascendante hiérarchique (CAH) 321
7.5. K-Means clustering (méthodes des centroides) 323
7.5.1. Méthodes des centroides fixes (fixer le nombre de clusters) 324
7.5.2. Méthodes des centroides mobiles (aide à la détection du nombre adéquat de clusters) 324
7.6. Aide à l'interprétation des clusters (projection des clusters sur des axes factoriels issus d'une ACP) 326
7.7. Classification par les méthodes Support Vector Machine (SVM) 329
7.8. Classification par les méthodes de Random Forest................ 331
7.9. Classification par les méthode des voisins les plus proches 332
7.9.1. Méthode des k-voisins les plus proches 333
7.9.2. Méthode des voisins proches dans un rayons 334
7.10. Recherche des valeurs optimales des paramètres dans les algorithmes (GRID SEARCH) 334
Annexe : Exercices et cas pratiques de programmation (résolus) 336
Exercices de programmation de base, de gestions de fchiers et de traitement de textes 336
Bibliographie 362

Chapitre 1 : Les bases de la programmation sous python

1.1. Introduction

Python est un langage de programmation généraliste fonctionnant dans une approche orientée-objet. Il offre un environnement complet de développement comprenant un interpréteur et un ensemble de bibliothèques. Il dispose d'un très large éventail de modules qui offre au programmeur des outils très divers pour différentes utilisations : écriture d'applications Web (Zope, Django), programmes de calculs scientifiques, élaboration d'interfaces graphiques, programmation de scripts systèmes, traitement de données, traitement de texte, analyse de données, gestion réseau (librairie socket), manipulation du format xml, accès aux protocoles d'Internet (protocoles des services courriel, divers protocoles web), accès aux éléments du système d'exploitation sous-jacent (accès aux fichiers et répertoires, gestion des processus), écriture d'interfaces graphiques (librairie Tkinter), accès aux bases de données relationnelles, etc.

Il existe plusieurs avantages à choisir python comme langage de programmation pour la Data Science. D'abord, il offre tous les avantages de la programmation orientée-objet et permet d'écrire un code facilement compréhensible. L'un des plus gros atouts du système Python est sa portabilité c'est-à-dire sa capacité à fonctionner sur différentes plates-formes (Mac OS X, Unix, Windows, etc). Cette portabilité permet l'exécution du même code sous différents environnements techniques. Python est un langage interprété (c'est-à-dire que chaque ligne de code est lue puis interprétée afin d'être exécutée) au lieu d'être un langage compilé. Par ailleurs son utilisation est parfaitement adaptée à l'intégration de composants écrit dans d'autres langages tels que C, C++ ou Java. De plus, le fait d'embarquer l'interpréteur python dans une application permet d'intégrer directement les scripts python au sein des programmes sans aucune réadaptation. Cette grande flexibilité et ainsi que cette capacité à s'intégrer à différents environnements techniques y compris les services web et les bases de données en font un langage de premier choix pour les analyses du Big Data.

Fonctionnement des programmes python

On dit généralement qu'«en Python, tout est objet ». Cette expression signifie que quels que soient les données que l'on manipule, celles-ci se présentent d'abord sous forme d'objet qui se défini comme une capsule contenant à la fois les attributs et les méthodes (ou fonctions associées). C'est le fondement de la programmation orientée-objet sous python

Le langage Python, comme la majorité des langages, peut être écrit aussi bien en mode interactif qu'en mode script. Dans le premier cas, il y a une interaction directe entre l'utilisateur et l'interpréteur python. Les commandes entrées par l'utilisateur sont évaluées au fur et à mesure que touche « Entrée » est tapée. Cette première solution est pratique pour exécuter les

lignes de commande simples ainsi que pour tester tout ou partie d'un programme. Pour une utilisation en mode script les instructions à évaluer par l'interpréteur sont sauvegardées dans un fichier enregistré avec l'extension .py. Dans ce second cas, l'utilisateur doit saisir l'intégralité des instructions qu'il souhaite voir évaluer à l'aide de son éditeur de texte, puis demander leur exécution à l'interpréteur.

Dans ce document, nous n'allons pas nous appesantir sur les détails concernant l'installation et la mise en route de python. Nous supposons alors que cette partie est déjà réalisée. Il existe de multiples tutoriels permettant une prise en main rapide de python. A noter simplement que ce présent document est réalisé en utilisant Python version 3.4 téléchargeable à partir du lien internet https://www.python.org/download/releases/3.4.0/

1.2. Les éléments structurants d'un programme Python

Tout programme python se construit autour d'un certain nombre d'éléments qui en constituent la structure. Il s'agit notamment de la définition des variables à travers un ensemble d'opérations arithmétiques ou logiques mais aussi de la définition des instructions de manière séquentielle. On distingue plusieurs types d'instructions : les instructions inconditionnelles (c'est-à-dire des instructions qui doivent toujours être exécutées chaque fois que le programme est lancé), les instructions conditionnelles (définies avec la condition if....) et les instructions en boucles (définies avec la clause Do while… et la clause for…). Le but de cette section est d'étudier chacun de ces éléments en donnant des détails sur leur définition et leur utilisations lors l'écriture d'un code python.

En plus des variables et des instructions, les programmes Pythons peuvent aussi contenir des commentaires de code. Ces commentaires s'avèrent incontournables notamment dans les programmes plus complexes permettant à d'autres utilisateurs de comprendre la logique d'écriture. Il est donc fortement recommandé de mettre autant de commentaires que nécessaires afin de faciliter la lecture du code. Les commentaires Python sont définis à l'aide du caractère #. D'une manière générale, tout écriture apparaissant à la suite du signe # sur une ligne est traitée par l'interpréteur comme un commentaire.

1.2.1. Les variables

1.2.1.1. Définir une variable

Sous Python, une variable est un objet de programmation permettant de stocker une information en mémoire de la machine en attribuant un nom symbolique. Une variable est définie en utilisant l'opérateur mathématique = soit par assignation directe de sa valeur, ou par une assignation à partir de la valeur d'une autre variable. Exemples :

1.2.1.1.1. Définir une variable par assignation directe

```
x = 7 # définit la variable nommée x et lui assigne la valeur 7
```

```
y = 3.458 # définit la variable nommée y et lui assigne la valeur 3.458
msg = "Comment allez-vous ?" #  Définit la variable nommée msg et lui assigne la valeur " Comment allez-vous ?"
```

Pour afficher les valeurs des trois variables définies, on utilise la fonction print()

```
print(x)
print(y)
print(msg)
```

Pour afficher les trois valeurs sur la même ligne on utilise une seule fonction print() en séparant les variables par des virgules :

```
print(x,y,msg)
```

NB : En mode interactif, il n'est pas nécessaire de préciser la fonction print(). Il suffit simplement de spécifier le nom de la variable et d'appuyer sur la touche Entrée. La valeur sera automatiquement affichée. Cette méthode raccourcie s'applique aussi à d'autres fonctions ou objets Python.

Le langage python offre plusieurs facilités pour réaliser les assignations de valeurs en raccourcissant les lignes de codes. On distingue par exemples les assignations multiples et les assignations parallèles.

Une assignation multiple consiste à attribuer une même valeur à plusieurs variables dans la même ligne de code. Exemple :

```
x = y = 7 # x et y prennent la même valeur simultanément 7.
```

Une assignation parallèle consiste à définir plusieurs variables en utilisant un seul symbole d'égalité. Exemple :

```
x, y = 4, 8.33 # On définit deux variables x et y dont les valeurs sont respectivement 4 et 8.33.
```

Nb : Les noms des variables (et les valeurs) sont séparées par les virgules de part et d'autre de l'égalité.

Pour fixer le nombre de décimaux lors de l'affichage d'un nombre, on utilise l'opérateur de formatage "%.nf"%nomVar où n est le nombre souhaité de décimaux à afficher et nomVar est le nom de la variable. Exemple : soit la variable x définie comme suit :

```
x=3.47568
```

On peut afficher la valeur de x avec un nombre de décimaux souhaitable comme suit :

```
print("%.2f"%x) # Affichage de x avec 2 chiffres après la virgule
```

```
print("%.3f"%x) # Affichage de x avec 3 chiffres après la virgule
print("%.0f"%x) # Affichage de x avec 0 chiffres après la virgule
```

Le symbole f signifie float qui représente un nombre réel avec possibilité de décimaux contrairement aux nombres entiers de symbole %i (nous reviendrons plus en détails sur les différents type des variables plus tard). Le symbole % est un opérateur de formatage de valeur. Il permet de faire apparaître la valeur d'une variable même au milieu d'une chaîne de caractères (nous reviendrons également sur les opérateurs de formatage de valeurs). L'expression %.2f"%x signifie que la variable x doit être traitée comme un nombre réel avec 2 chiffres après la virgule. On peut mettre autant de valeurs qu'on souhaite. Par exemple, en mettant 0, on retombe sur le cas d'un nombre entier (aucun chiffre après la virgule). Ce qui peut être traduit en utilisant la fonction %i comme suit :

```
print("%.i"%x) # Affichage de x comme un entier naturel (integer)
```

Par ailleurs pour traiter x comme une chaine de caractères, on utilise la fonction %s (formatage en string). Ainsi, on a :

```
print("%.s"%x) # Affichage de x comme une chaîne de caractères
```

1.2.1.1.2. Définir une variable à partir d'autres variables

```
z1=x+y # définit la variable nommée z1 et lui assigne la somme des variables x et y
z2=x+5 # définit la variable nommée z2 ajoutant 5 à la valeur de x
z3=2*y # définit la variable nommée z3 en multipliant la valeur de y par 2
```

NB : La définition des noms des variables obéissent à des règles bien précises. En effet, un nom de variable doit débuter par une lettre ou par le caractère de soulignement (underscore _) suivi par un nombre quelconque de lettres, chiffres ou de caractères de soulignement. L'underscore est le seul caractère spécial autorisé. Par exemple, le tiret du 6 n'est jamais autorisé dans la définition d'un nom de variable.

Par ailleurs, il est conventionnel d'écrire les noms des variable en minuscule (par exemple x au lieu de X, var1 au lieu de VAR1). Lorsque la variable doit avoir un nom composée (ex : ma variable), il est préférable de commencer la spécification des autres éléments du nom par un majuscule. Ex : maVariable, testValue, etc.. Cela facilite beaucoup la compréhension d'un programme. Tout comme pour les variables, les noms des fonctions doivent commencer par des minuscules. Mais les lettres en majuscule peuvent être autorisées pour des noms composés afin de faciliter la lecture du nom. Les noms commençant par une majuscule sont utilisés pour la définition des objets de types classe (nous y reviendrons).

1.2.1.1.3. Définir une variable à partir d'une séquence de valeurs

Les séquences de valeurs sont des variables très fréquemment rencontrées dans les programmes python. Elles correspondent à un ensemble de valeurs successives et ordonnées

dont les valeurs peuvent être extraites comme une liste. Les séquences de valeurs sont générées en utilisant la fonction range(). Exemple :

```
x=range(10)  # Crée une séquence de valeurs entières allant de 0 à 9
x=range(2, 10)  # Crée une séquence de valeurs entières allant de 2 à 9
x=range(1, 10, 2)  # Crée une séquence de valeurs entières allant de 2 à 9 avec un pas de 2. Il renvoie alors 1, 3, 5, 7 et 9
```

Pour afficher les valeurs générées, on peut utiliser la fonction list() ou la fonction tuple() telles que :

```
x=range(10)
print(list(x)) # renvoie une liste :valeurs entre crochets [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(tuple(x)) # renvoie un tuple : valeurs entre parenthèses (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Nous reviendrons plus en détails sur la définition des objets liste ou tuple qui correspondent chacun à un objet particulier dans Python.

1.2.1.2. Type d'une variable

Définir le type d'une variable

Le type d'une variable correspond à la nature de celle-ci. Il existe de nombreux autres types de variables (nombre entier, nombre réel, chaînes de caractères, etc). Les types de variables les plus couramment rencontrés sont les entier (int) , les nombres réels (float), les chaînes de caractères (str). A chacun de ces types de variables est associé un certain nombre d'attributs ou de méthodes. Par exemple, pour deux nombres 20 et 45. En faisant 20+45, on obtient 65. Par contre pour deux caractère A et B, en faisant "A"+"B" on obtient "AB"

Python est un langage au typage dynamique, c'est-à-dire qu'il reconnaît directement le type des variables lorsque sa valeur est indiquée. Ainsi, contrairement à d'autres langages comme le langage C, il n'y a pas de déclaration de type obligatoire lors de la création d'une variable. Exemples :

```
x = 2 # type entier (int)
x =2. # type réel (float avec décimaux)
x='2' # type caractère (string)
```

Pour connaitre le type d'une variable, on utilise la fonction type(). Par exemples :

Pour la définition x=2, en faisant type(x) renvoie <class 'int'> signifiant un entier (integer). Pour la définition x=2., la fonction type(x) renvoie <class 'float'> signifiant un nombre réel. Et pour la définition x='2', la fonction type(x) renvoie <class 'str'> signifiant que x est de type chaîne de caractères (string).

NB : Il faut remarquer que les variables en chaînes de caractères sont toujours définies avec des guillemets (qui peuvent être des guillemets simples, doubles ou triples). Avec des guillemets simples, on a des chaines de types char alors qu'avec des guillemets doubles, on obtient des chaînes de type string. Noter aussi que même si une variable est définie avec des nombres, lorsque ces nombres sont indiqués avec des guillemets, python traite cette variable comme une chaine de caractères. Néanmoins, il existe des fonctions pour convertir une variable d'un type à un autre lorsque cette conversion est autorisée. Voir les exemples suivants.

Tester le type d'une variable

Exemple : soient les variables i, j,k,m

```
i = 123
type(i) # renvoie <type 'int'>
type(i) is int  # renvoie True
j = 123456789L
type(j) # renvoie <type 'long'>
type(j) is long  # renvoie True
k = 123.456
type(k) # renvoie <type 'float'>
type(k) is float # renvoie True
m = 'ABCD'
type(m) # renvoie <type 'str'>
type(m) is str # renvoie True
```

Convertir le type d'une variable

Exemple 1 : Conversion d'une variable de type caractère en variable de type entier ou réel.

Soit x='2' une variable en caractère définie par 2. Pour convertir cette variable en type numérique (entier), on utilise la fonction int(). On a :

```
x=int(x)
```

Pour convertir en un nombre réel autorisant les décimaux, on fait :

```
x=float(x)
```

NB : La conversion d'une variable de type caractère en variable de type numérique n'est possible que lorsque la valeur de la variable est constituée uniquement de chiffres. La conversion renvoyera un message d'erreur lorsque la valeur de x contient une lettre. Exemple : pour x='2xeb'. En essayant de convertir cette variable en type numérique int, on reçoit un message d'erreur « invalid literal for int() with base 10: '2xeb ' ». Et avec la fonction float, on reçoit le message « could not convert string to float: '2xeb' ».

Exemple 2 : Conversion d'une variable de type numérique (entier ou réel) en variable de type caractère.

Soit x=2 une variable numérique de type entier définie par 2. Pour convertir cette variable en type caractère, on utilise la fonction str(). On a :

```
x=str(x)
```

On utilise aussi la même fonction lorsqu'il s'agit d'une variable numérique de type réel comme par exemple x=2.4756.

```
x=str(x)
```

En conclusion, on peut noter que toutes les variables numériques sont convertible en type caractère mais une variable n'est convertible en type numérique que lorsque la valeur de celle-ci est constituée uniquement de chiffres. Toutefois, les conversions sont possibles vers d'autres formats de données comme les nombres du système hexadécimal.

Manipulation du type d'une variable dans un dataframe

Pour connaitre le type d'une variable dans un dataframe, on utilise la fonction dtypes. Exemple :

```
mydata.dtypes # renvoie le type de chaque variable présente dans la table de données mydata.
mydata['myvar1'] # renvoie le type de la variable myvar1 de la table mydata
mydata['myvar1', 'myvar2'] # renvoie le type des deux variables myvar1 et myvar2
```

NB : Les variables en caractères sont représentées par le type « object », les variable numérique entières par « int », les variables numérques décimales par « float ».

1.2.1.3. Méthodes associées aux variables

A chaque variable créée dans python, est associé un ensemble d'attributs (ex : type) mais aussi un ensemble de méthodes, c'est à dire un ensemble d'opérations de traitement et d'exploitation réalisable avec cette variable. Pour afficher l'ensemble des méthodes d'une variable, on utilise la commande dir(). Exemple :

```
x=2.5 # Définit une variable numérique x
y='mon texte' # Définit une variable en chaîne de caractères y.
```

Pour afficher l'ensemble des méthodes associées à chacune de ces variables, on fait :

```
print(dir(x))
print(dir(y))
```

Ainsi, pour obtenir de l'aide sur une méthode spécifique, on utilise la fonction help() comme suit : help(x.nomMethode) # où nomMethode est le nom de la méthode considérée. Par

exemple, pour une variable numérique de type float, il existe une méthode nommé conjugate. Pour obtenir de l'aide sur cette fonction, on fait :

```
print(help(x.conjugate))
```

Pour afficher l'aide sur toutes les fonctions associées à la variable x, on fait simplement :

```
print(help(x))
```

1.2.1.3. Les opérateurs arithmétiques et logiques (booléens)

Deux grandes catégories d'opérateurs sont utilisées en Python pour définir les variables et les instructions. Il s'agit des opérateurs arithmétiques et algébriques et les opérateurs logiques. Les premiers permettent de réaliser les opérations mathématiques courantes tandis que les seconds permettent de réaliser des comparaisons de valeurs. Une valeur booléenne est une évaluation logique représentant l'une des deux possibilités suivantes : vrai ou faux. Les valeurs booléennes sont le résultat de l'évaluation d'expressions logiques et elles servent à faire des choix dans un programme (effectuer telle action quand telle condition est réalisée).

Le tableau ci-dessous fournit les détails sur les différents opérateurs en Python. Pour accéder à la liste complète des opérateurs standards de python et leurs équivalents fonctions, voir la page https://docs.python.org/2/library/operator.html. Consulter http://www.tutorialspoint.com/python/python_basic_operators.htm pour quelques exemples d'utilisations des opérateurs standards.

Opérateurs arithmétiques

	Symbole	Exemple
Addition	+	x=2+3
Soustraction	-	z=x-y
multiplication	*	y=3*x
Division (quotient réel)	/	z=5/2 # renvoie 2.5
Division (quotient entier)	//	z=5//2 # renvoie 2
Puissance	**	x**2 # x puissance 2
Reste de la division (modulo)	%	17%3 # renvoie 2
Addition incrémentée	+=	x+=4 # ajoute 4 à la valeur initiale de x

		(équivalent à x=x+4)
Soustraction incrémentée	-+	x-=4 # soustrait 4 de la valeur initiale de x (équivalent à x=x-4)
Opérateurs logiques (variables booléennes et valeurs tests)		
Egal	==	x==2 # Egalité logique ou mathématique (différent de l'égalité d'assignation =)
Strictement inférieur	<	x<2
Inférieur ou égal	<=	x<=2
Strictement supérieur	>	x>2
Supérieur ou égal	>=	x>=2
Différent	!=	x !=2
Et	and (équivalent à &)	x>2 and x< 10 # expression logique composée
Ou	Or (équivalent à |)	x==2 or x==2 # Expression logique composée
Opérateur de négation	not (équivalent à ~)	x not True

Nb : Toute expression logique renvoie deux valeurs (True ou False). La valeur True est renvoyée lorsque l'expression est vérifiée. Dans le cas contraire, elle renvoie False. Ces deux valeurs sont qualifiées de valeurs booléennes. Exemple : Soit la variable note définie comme suit :

```
note = 13.0
```

On va tester quelques expressions logiques :

```
print(note==13.0) #  Renvoie True
print(note >= 12.0 and note < 14.0)  #  Renvoie True
```

```
print(note<50) # renvoie True
print(note>20) # renvoie False
print(note) # renvoie False
print(not note>20) # renvoie True
```

1.2.2. Les instructions conditionnelles : la clause « if… else »

Les instructions conditionnelles sont des opérations que le programme doit exécuter lorsqu'il expression logiques renvoie True ou False (selon le cas). Les instructions conditionnelles sont définies à l'intérieur d'une clause if…else

La structure générale d'un bloc if…else se présente comme suit :

```
if expression_logique :
      instruction 1
      instruction 2
      ..
      instruction n
else :
      autre_bloc_instructions
```

Attention au symbole « : » à la suite du mot clé if ou else qui marque la fin de la déclaration de la fonction if ou else et le début de la définition des instructions.

Noter aussi que la clause else n'est pas obligatoire c'est-à-dire qu'il peut y avoir une clause if sans clause else.

Voici ci-dessous quelques exemples d'utilisation de la fonction if.

Exemple 1 : Instructions avec une clause « if » simple

```
x = 150
if (x > 100):
    print("x dépasse 100")
```

On définit d'abord une variable x en lui assignant la valeur 150. Ensuite on définit une expression logique permettant de vérifier si x est supérieur à 100 ou pas. Dans cette expression conditionnelle, lorsque l'expression logique est vraie (c'est-à-dire renvoie True), on affiche un message indiquant que x est supérieur à 100. Lorsque la condition n'est vérifiée aucun message ne sera affiché. Il faut tout de même signaler ici que puisque la valeur de x est 150 (donc supérieur à 100) alors la condition sera toujours vérifiée et le message sera tours affichée. Cela jusqu'à ce qu'on assigne une nouvelle valeur à x qui soit inférieur à 100.

NB : Dans une clause if, les instructions (ici print()) doivent être espacées de 4 pas (avec la touche tab), de manière à ce que celle-ci soit indentée (c'est-à-dire en retrait par rapport à la clause if à laquelle elle est rattachée). En effet, dans le langage Python, les blocs d'instructions

sont définis grâce à la suite d'un décalage (ou indentation). Contrairement à, d'autres langages, comme le C qui utilisent des marqueurs plus explicites {bloc d'instructions }. L'avantage de rendre le décalage obligatoire comme marqueur de bloc d'instructions est qu'il oblige à une écriture claire des programmes.

Il faut aussi noter que les parenthèses utilisées avec l'instruction if sont optionnelles. Elles sont simplement utilisées pour améliorer la lisibilité de l'expression logique (qui peut être très complexe dans certains cas).

Exemple 2 : Instructions avec la clause if… else…

```
x = 150
if (x > 100):
    print("x dépasse 100")
else:
    print("x ne dépasse pas 100")
```

Contrairement au premier exemple où il n'y avait pas d'instructions alternative à exécuter dans le cas où l'expression logique n'est pas vérifiée, dans ce deuxième exemple, on définit une instruction alternative en utilise la clause else…

```
x = 7
if (x % 2 == 0):
    print("x est pair car le reste de sa division par 2 est nul")
else:
    print("x est impair car le reste de sa division par 2 n'est pas
nul ")
```

Exemple 3: Instructions avec la clause if… elif… else…

La clause if… elif… else est utilisées lorsqu'on dispose de plusieurs conditions logiques auxquelles sont associées des instructions à exécuter lorsqu'elles sont vérifiées.

```
x = 0
if x > 0 :
    print("x est positif")
elif x < 0 :
    print("x est négatif")
else:
    print("x est nul")
```

Nb : La clause elif peut être utilisée autant de fois qu'il y a de conditions alternatives entre la clause initiale if et la clause finale else. Toutefois, l'utilisation de la clause elif n'est pas obligatoire pour traduire les conditions alternatives. On peut simplement utiliser la clause if à la place.

Le résultat d'une expression logique peut être placé dans une variable booléenne. Une telle variable prend alors deux valeurs booléennes possibles (True ou False). Par exemple :

```
condition = (x >=0 and x <50)
if condition :
      print ("La condition est vérifiée")
if not condition :
      print ("La condition n'est pas vérifiée")
```

Exemple 4 : Définir des clauses « if » imbriquées

Les exemples précédents traduisent des clause de premier niveau. Dans les programmes complexes, il arrive très fréquemment que les clauses if soient imbriquées. C'est-à-dire que des clauses if sont définies à l'intérieur d'autres clauses if. Et cela à plusieurs niveaux. Voir exemple ci-dessous :

```
embranchement="rien"
if embranchement == "vertébrés":
    if classe == "mammifères":
        if ordre == "carnivores":
            if famille == "félins":
                print("Il s'agit peut-être d'un chat")
        print("c'est en tous cas un mammifère")
    elif classe == "oiseaux":
        print("c'est peut-être un canari")
print("la classification des animaux est complexe")
```

1.2.3. Les instructions en boucle

Sous python, on distingue deux principales catégories d'instructions en boucles : les instructions en boucle « while… » et les instructions en boucle « for… in… »

Les instructions en boucle « while… » sont exécutées tant que la (les) condition(s) définies par l'expression logiques sont vraies. Tandis que les instructions en boucle « for…in » sont exécutées pour chaque élément pris dans un ensemble d'éléments préalablement indiquées (généralement dans un objet range ou list).

1.2.3.1. Les instructions en boucle « while… »

La syntaxe générale de la définition des instructions en boucle while se présente comme suit :

```
Initialisation de la variable d'incrémentation
while condition :
      bloc_instructions
      incrémentation
```

Une boucle while nécessite trois éléments pour fonctionner correctement : 1. l'initialisation de la variable de test (ou variable d'incrémentation) avant la boucle ; 2. La définition de la condition à évaluer définie en fonction de la variable de test; 3. l'incrémentation de la variable de test dans le corps de la boucle à la suite des instructions.

Là aussi, il faut prêter attention au symbole « : » qui marque la fin de la déclaration de la boucle while et le début de la définition des instructions tout comme dans le cas de l'utilisation de la fonction « if »

L'exemple ci-dessous illustre la définition d'une boucle while… :

```
x = 1  # Initialisation de la variable x
while (x < 10):
     print('La valeur de x est ', a)
     x = x + 1 # Incrémentation de la variable x
```

Dans l'exemple ci-dessus, on affiche la valeur de x tant que x est inférieur à 10. La valeur de x est initialisée à 1 et incrément avec +1 après chaque instruction print(). Comme dans la structure if, la condition est d'abord évaluée et si elle vraie, le bloc d'instructions est exécuté. Mais ici avec la clause while après exécution du bloc d'instruction, la condition est évaluée à nouveau. Cette exécution se répète jusqu'à ce que la condition devienne fausse. C'est pourquoi il n'est nécessaire de définir une variable d'incrémentation dont la valeur se modifie après chaque exécution de sorte que la condition puisse être fausse à partir d'une certaine valeur. Cette incrémentation est nécessaire pour éviter que les instructions soient indéfiniment évaluées créant ainsi une boucle infinie (ou boucle folle). Une telle situation nécessite alors un arrêt forcé de l'exécution du programme. Dans ce cas, on peut utiliser la touche « CTRL+C » pour interrompre l'exécution du programme.

NB : Il faut aussi noter que comme dans la définition des instructions en clause if, les blocs d'instruction dans une clause while sont aussi définis par indentation (décalage de positions à l'intérieur de la clause while)

1.2.3.2. Les instructions en boucle « for… in… »

La syntaxe générale de la définition d'instructions en boucle for… in… est la suivante :

```
for element in sequence_de_valeurs :
     bloc instructions
```

Dans cette syntaxe, le bloc d'instruction est exécuté pour chaque élément figurant dans la séquence de valeurs qui est généralement une liste de valeurs ou un tuple de valeurs (nous reviendrons sur la définition des listes et de tuples un peu plus tard).

Contrairement à la variable d'incrémentation dans une boucle while, la variable indice (element) n'est pas initialisée par l'utilisateur. Elle prend successivement les valeurs figurant dans la séquence parcourue.

Nb : element est ici un nom générique, on peut utiliser n'importe quel nom à la place.

Là également, il faut prêter une attention au symbole « : » qui marque la fin de la déclaration de la boucle « for » et le début de la définition des instructions tout comme dans le cas de l'utilisation de la fonction « if » et de la boucle « while »

Les exemples ci-dessous illustrent la définition d'une boucle « for ».

Exemple 1 : Instruction dans une boucle « for.. in ... » simple

```
for i in range(1,11) :
print(i)
```

Dans cet exemple, on affiche chaque valeur défini par la séquence de valeur range(1, 11) qui fournit en fait les valeurs allant de 1 à 10. On pouvait aussi utiliser la structure syntaxique suivante :

```
for i in [1,2,3,4,5,6,7,8,9,10]:
print(i)
```

Ce qui vaut en réalité à la commande suivante :

```
for i in list(range(1,11)):
print(i)
```

Exemple 2 : Boucle « for.. in... » sur une chaine de caractères

```
listch = "Hello world"
for i in listch :
    print ( i )
```

Attention : Dans cet exemple, il ne s'agit pas de répéter les instructions pour les deux mots mais plutôt pour chaque par lettre apparaissant dans la séquence de caractères y compris les espace. C'est la particularité des chaînes de caractères. Nous reviendrons plus en détails sur les opérations sur les chaînes de caractères plus tard.

Exemple 3 : Combinaison d'une boucle « for...in ... » et d'une clause « if »

```
listnb = [4, 5, 6]
for i in listnb:
    if i == 5:
        print("La condition est vérifiée pour l'élement", i)
    else :
        print("La condition ,n'est pas vérifiée pour l'élément", i)
```

```
mych = "Hello World"
for i in mych :
    if i in "AEIOUYaeiouy":
         print ('La lettre', i, 'est une voyelle')
    else :
        print ('La lettre', i, 'est une consonne')
```

1.2.4. Les instructions break et continue dans une clause « while… » ou « for … in…»

Les mots réservés break et continue sont utilisés pour modifier le comportement d'une boucle « for…in » ou d'une boucle « while… ». L'instruction *break* permet d'arrêter l'évaluation de la boucle et l'exécution des instructions pour sortir prématurément de la boucle même la condition principale définissant la boucle reste encore vraie. Et l'instruction *continue* permet de suspendre l'exécution des instructions lorsque la condition principale est vérifiée. Les exemples ci-dessous sont des illustrations.

Exemple de boucle avec l'instruction break

```
for i in range(5):
    if i > 2:
        break
    print (i)
```

Au départ, cette boucle for vise à afficher tous éléments de la séquence de valeurs allant de 1 à 4. Mais avec l'instruction break définie à l'intérieur de la clause if, la boucle for est stoppée lorsque i devient supérieur à 2.

Exemple de boucle avec l'instruction continue

```
for i in range(5):
    if i == 2:
        continue
    print (i)
```

Au départ, cette boucle for est conçue pour afficher tous éléments de la séquence de valeurs allant de 1 à 4. Mais avec l'instruction *continue* définie à l'intérieur de la clause if, l'instruction print() n'est pas exécutée lorsque i prend la valeur 2. Mais elle est exécutée pour toutes les autres valeurs de la séquence.

1.2.5. Les mots réservés du langage python

Comme dans tout langage de programmation, il existe des mots clés réservés qui font partie du socle du langage. Ces mots réservés (qui sont pour la plupart des fonctions) ne peuvent pas

être modifiés par l'utilisateur, et leur dénomination ne peut pas être utilisée pour former le nom d'une variable, d'une fonction ou tout autre objet définis par l'utilisateur.

Sous Python, il existe 33 mots réservés dont la liste est fourni ci-dessus. Certain de ces mots clés seront sollicités à de nombreuses reprises dans nos discussions dans ce document.

and	elif	if	or	yield
as	else	import	pass	
assert	except	in	raise	
break	False	is	return	
class	finally	lambda	True	
continue	for	None	try	
def	from	nonlocal	while	
del	global	not	with	

1.3. Les principaux objets du langage Python

Comme signalé dans les sections précédentes, un programme python fonctionne toujours sur la base des manipulations des objets. Les instructions (qu'elles soient définies dans un code libre, dans une clause if ou dans une boucle while ou for) sont toujours exécutées sur des objets. Les variables telles que discutées précédemment représentent un type particulier d'objet python. Sinon d'une manière générale, les instructions dans un programme python sont définies à partir d'une collection d'objets qui se présentent sous différentes formes de séquences de valeurs. Cette séquence de valeurs peut être par exemple constituée par un ensemble de variables à valeur unique. Ce qui signifie finalement qu'une variable définie à partir d'une seule valeur n'est généralement pas suffisant pour former l'architecture d'un programme. En effet, sous python, les variables sont généralement des objets qui prennent plusieurs valeurs. Le vecteur ou une matrice sont illustrations qui peuvent aider à comprendre la notion de collection de valeurs. Très concrètement, les principaux objets de programmation de python sont les listes, les tuples, les ensembles (set), les tableaux (array), les dictionnaires, les fonctions, les classes et les modules. Nous allons étudier plus amplement chacun de ces types d'objets dans ce chapitre. Dans cette section, nous nous limiterons d'abord à une présentation sommaire de chacun des objets. En effet, dans le langage python :

- Un objet **liste** est une séquence de valeurs (numériques et /ou caractères) **indicées** et spécifiées à l'intérieur des **crochets**, séparées par des virgules. Exemple :

```
x=[ 1 , 5 , 8, 10 ,4 , 50, 8 ] # Liste formée uniquement de chiffres
y=["Olivier","ENGEL","Strasbourg"] # liste formée uniquement de caractères
z=[1, "Olivier",5 , 8, "ENGEL",10, 4, 50, "Strasbourg"] # liste formée de chiffres et de caractères.
```

Un objet liste peut être généré manuellement en indiquant les valeurs entre crochets ou automatiquement en utilisant la fonction list()

```
x=list(range(1,10))
print(x) # renvoie [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Un objet **tuple** est une séquence de valeurs (numériques et /ou caractères) **indicées** et spécifiées à l'intérieur des **parenthèses**, séparées par des virgules. Exemple :

```
x=[ 1 , 5 , 8, 10 ,4 , 50, 8 ] # Tuple formé uniquement de chiffres
y=["Olivier","ENGEL","Strasbourg"] # Tuple formé uniquement de caractères
z=[1, "Olivier",5 , 8, "ENGEL",10, 4, 50, "Strasbourg"] # Tuple formé de chiffres et de caractères.
```

Un objet tuple peut être généré manuellement en indiquant les valeurs entre parenthèses ou automatiquement en utilisant la fonction tuple(). Exemple :

```
x=tuple(range(1,10))
print(x) # renvoie (1, 2, 3, 4, 5, 6, 7, 8, 9)
```

- Un objet **set** est une séquence de valeurs (numériques et /ou caractères) **non dupliquées** et **non indicées,** et spécifiées à l'intérieur des **accolades**, séparées par des virgules. Exemple :

```
x={ 1 , 5 , 8, 10 ,4 , 50 } # Set formée uniquement de chiffres
y={"Olivier","ENGEL","Strasbourg"} # Set formée uniquement de caractères
z={1, "Olivier",5 , 8, "ENGEL",10, 4, 50, "Strasbourg"} # Set formé de chiffres et de caractères
```

Un objet set peut être généré manuellement en indiquant les valeurs entre accolades ou automatiquement en utilisant la fonction set()

```
v=[2 , 4 , "orange", "viande", 4, "orange"]
x=set(v)
print(x) # renvoie {2, 'orange', 4, 'viande'}
```

- Un objet **array** est une séquence de valeurs se présentant sous forme de tableaux (avec des lignes et des colonnes) et exclusivement constituées de valeurs **numériques indicées** et spécifiées à l'intérieur des **crochets**, séparées par des espaces. Exemple :

```
x=[ 1  5   8  10  4   50  8 ] # Array à une dimension
y=[ [1  5   8]  [10  4 ]  [50  8] ] # Array à deux dimensions (array de array)
z=[ [ [1  5]   [8  10]]  [[4   50]  8] ] # Array à trois dimensions (array de array de array)
```

Attention : Le terme array à plusieurs dimensions ne signifie pas array à plusieurs colonnes. Il signifie array de array, ainsi de suite.
On peut générer un objet array manuellement en spécifiant les valeurs comme indiquées ci-dessus. On peut aussi utiliser la fonction numpy.array(). Il faut pour cela importer le module numpy en utilisant la commande :

```
import numpy
```

Nous reviendrons plus en détails sur la définition des arrays.

- Un objet **dictionnaire** est une séquence de valeurs (numériques et /ou caractères) **indicées** par des clés et spécifiées à l'intérieur des **accolades** et séparées par des virgules**.** A chaque clé correspond une ou plusieurs valeurs. Un dictionnaire est constitué d'un ensemble clé-valeurs. Exemple :

```
x = {'nom':'Jean', 'poids':70, 'taille':1.75} # Clés à valeurs uniques.

y={'Jean':[25,70,1.75],'Paul':[30,65,1.80],'Pierre':[35,75,1.65]}
# Clé à plusieurs valeurs formée par une liste
```

Pour le dictionnaire x, les clés sont nom, poids et taille. Les valeurs correspondant à chacun de ces clés sont respectivement jean, 70 et 1.75.
Pour le dictionnaire y, les clés sont Jean, Paul et Pierre alors que les valeurs sont des listes constituées de trois valeurs (âge, poids et taille).
Un objet dictionnaire peut être généré manuellement en indiquant les valeurs entre accolades ou automatiquement en utilisant la fonction dict() pour créer un dictionnaire vide. Nous reviendrons plus en détails sur la définition des dictionnaires.

- Un objet **fonction** est un programme conçu pour réaliser une opération bien précise. Par exemple la fonction print() permet d'afficher à l'écran les résultats de l'instruction qu'on lui fournit.
- Un objet **classe** est une collection de fonctions c'est-à-dire une association de fonctions apparentées
- Un objet **module** est une collection de classe c'est-à-dire une collection de classes apparentées

1.4. Etude des objets « liste »

1.4.1. Caractéristiques générales des listes

Comme indiqué dans la section précédente, un objet liste est une variable python définie à partir d'une séquence de valeurs (numériques et/ ou en caractères) indicées, spécifiées à

l'intérieur d'un crochet et séparées par des virgules. Cette définition, sans doute, descriptive et directe est formulée à partir de quelques mots clés qui méritent un peu de commentaires.

D'abord, un objet liste est une séquence de valeurs. Ce qui signifie qu'une liste est nécessairement un ensemble de valeurs (au moins une) même si théorique, il peut définir une liste vide.

Les valeurs qui constituent une liste peuvent être soit des chiffres, soit des chaines de caractères ou une combinaison des deux. Cette propriété des listes est très intéressante car elle permet d'aller au-delà de la définition traditionnelle du type d'une variable (variable numérique, variable caractère, etc). Une liste est donc d'un type plus général.

Les valeurs d'une liste sont indicées. En d'autres termes chaque valeur de la liste peut être identifiée à travers son indice c'est-à-dire sa position dans l'ordre des valeurs étant donné qu'une liste est un ensemble ordonné. En effet, la première valeur d'une liste a pour indice 0, la seconde valeur 1, ..., la dernière valeur a pour indice n-1 où n est le nombre total d'éléments de la liste.

1.4.1.1. Définition d'un objet liste

Un objet liste peut être déclaré et défini manuellement ou en utilisant la fonction list(). On distingue deux catégories de listes. Les listes à une dimension (ou listes simples) et les listes à plusieurs dimensions.

1.4.1.2.1. Définition d'une liste simple

Une liste simple est une liste dont les éléments sont constitués de valeurs uniques séparées par des virgules. Exemple :

```
x=['lundi','mardi','mercredi',1800,20.357,'jeudi','vendredi']
```

La variable x est une liste simple car les éléments sont des valeurs uniques, lundi, mardi, mercredi, 1800, ...,

Il faut remarquer que les éléments d'une liste peuvent être de types variés (chiffres et caractères). Pour la liste x, on constate que les trois premiers éléments sont des chaînes de caractères, le quatrième élément est un entier, le cinquième un réel. Et les deux derniers éléments sont des caractères. Les chiffres (entiers et réels) sont écrits simplement alors que les caractères sont toujours écrits entre les guillemets (simples, doubles ou triples).

Il faut noter aussi qu'on peut créer une liste en utilisant la fonction liste. Exemple :

```
x=list(['lundi','mardi','mercredi',1800,20.357,'jeudi','vendredi'])   # défintion inéfficiente car réalisée déjà sur une liste
y=list(range(10)) # Liste formée par une séquence de nombres entier de 0 à 9
```

Création d'une liste vide

Par ailleurs, dans certaines situations, il arrive qu'on crée d'bord une liste vide et qui sera par la suite remplie par des valeurs en utilisant la fonction append(). Pour créer une liste vide, on fait :

```
x=list()
```

Ou bien

```
x=[]
```

1.4.1.2.3. Définition d'une liste à plusieurs dimensions

Contrairement à une liste simple, une liste à plusieurs dimensions est une liste dont les éléments individuels sont constitués de plusieurs valeurs. D'une manière générale, une liste à plusieurs dimensions se présente comme une liste de listes. Les deux exemples ci-dessous sont des illustrations.

```
 x=[[1,2,3],[2,3,4],[3,4,5]]  #  liste  à  deux  dimensions  (liste  de
listes)
y= [[[1,2],[2,3]],[[4,5],[5,6]]] # liste à trois dimensions (liste de
listes de listes)
```

1.4.1.2. Indiçage d'une liste (slicing)

1.4.1.2.2. Indiçage d'une liste simple

La liste étant une séquence de valeurs indicées, on peut accéder à chacun des valeurs en indiquant leur indice. Exemple :

```
x=list(['lundi','mardi','mercredi',1800,20.357,'jeudi','vendredi'])    #    Définition
d'une liste

print(x) # affiche tous les éléments de la liste x
x[0] # renvoie le premier élément de x :lundi (Nb :l'indiçage commence
toujours à 0)
x[3] # renvoie l'élément d'indice 3( quatrième élément de x) :1800
x[1:3] # Renvoie tous éléments compris entre l'indice 1 et l'indice 3
(Nb : l'élément d'indice 3 est exclu)
x[1:6 :2] # Renvoie tous éléments compris entre l'indice 1 et l'indice
6 avec un saut de 2 éléments à chaque fois ['mardi', 1800, 'jeudi']
(l'élément d'indice 6 est exclu).
x[2 :] # renvoie tous éléments à partir de l'élément d'indice 2 (
inclu).
x[:3] # renvoie tous éléments situés avant l'élément d'indice 3
(exclu)
x[-1] # Indiçage négatif, renvoie le dernier élément de la liste
(équivaut ici à x[6])
```

```
x[-2] # Indiçage négatif, renvoie l'avant-dernier élément de la liste(équivaut ici à x[5])

x[: :2] # Parcourt tous éléments compris entre l'indice 0 et le dernier indice en renvoyant tous les éléments avec un saut de 2 éléments à chaque fois ['lundi', 'mercredi', 20.357, 'vendredi'].

x[: :-1] # renvoie une liste contenant tous éléments de x en les réorganisant du dernier élement au premier élément. Il s'agit de faire reverse sur x. renvoie ['vendredi', 'jeudi', 20.357, 1800, 'mercredi', 'mardi', 'lundi']. On obient le même résultats en faisant x.reverse()
x_rev=x.reverse()
```

Nb : En règle générale dans les slicing tout comme dans les séquences range(), les borne supérieures sont exclues des valeurs renvoyées.

1.4.1.2.4. Indiçage d'une liste à plusieurs dimensions (slicing à plusieurs niveaux)

Avec une liste à plusieurs dimensions, l'indiça se fait à plusieurs niveaux. Par exemple pour une liste à deux dimensions, l'indiçage de premier niveau permet d'accéder aux listes qui forment la liste principale (les élément-listes). Et l'indiçage de second niveau permet d'accéder aux éléments qui forment les élément-listes. Exemple :

Soit la liste x définie comme suit :

```
x=[[1,2,3],[2,3,4],[3,4,5]]
x[0] # renvoie le premier élément-liste  [1,2,3]
x[0][0] # renvoie le premier élément du premier élément-liste  [1,2,3] c'est-à-dire 1
x[2] # renvoie [ 3,4,5]
x[2][1] # renvoie 2
x[1:] # renvoie [[2, 3, 4], [3, 4, 5]]
x[1:][0] # renvoie [2, 3, 4]
x[-1] # renvoie 3, 4, 5]
x[1][:2] # renvoie [2, 3]
x[1][1:] # renvoie [3, 4]
```

1.4.3. Les opérations sur les listes

Une liste étant définie, on peut réaliser plusieurs opérations visant à modifier la structure de la liste ou les éléments qui le constituent. Dans cette section, nous passons en revue les opérations couramment réalisées sur les listes.

1.4.3.1. Déterminer le nombre d'éléments d'une liste

Pour connaitre le nombre d'éléments d'une liste, on utilise la fonction len(). Exemple : soit la liste x définie comme suit :

```
x= ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
```

Pour connaitre la longueur de x (nombre d'éléments de x), on fait :

```
print(len(x)) # cela renvoie 7
```

Nb : Il est possible de stocker cette valeur dans une variable pour des utilisations futures. Par exemple dans l'élaboration des boucles for utilisant les indices des éléments. Pour récupérer la valeur de len(x), on l'assigne directement à une variable. Ex :

```
l=len(x)
print(l)
```

1.4.3.2. Additionner des listes

On peut additionner deux listes pour en former une seule en utilisant l'opérateur + qui permet de concaténer les listes en questions Exemples : soient les deux listes x et y définies par :

```
x= ['girafe','tigre']
y= ['singe','souris']
```

On peut créer une liste z en faisant la somme des deux listes telle que :

```
z= x + y
print(z) # renvoie  ['girafe', 'tigre', 'singe', 'souris']
```

L'opérateur + permet d'adjoindre les éléments de la liste y à ceux de la liste x pour former la liste z.

1.4.3.3. Démultiplication des éléments d'une liste

On peut démultiplier les éléments d'une liste en utilisant l'opérateur de multiplication * en indiquant un entier correspondant au nombre fois qu'on souhaite démultiplier les éléments. Exemple :

Soit la liste x définie comme suit :

```
x=['girafe', 24, 18 , 'tigre', 2400, 150 ]
```

On souhaite démultiplier tous les éléments de x par 3 pour former une liste y. On a :

```
y=x*3
print(y)  # renvoie ['girafe', 24, 18, 'tigre', 2400, 150, 'girafe',
24, 18, 'tigre', 2400, 150, 'girafe', 24, 18, 'tigre', 2400, 150]
```

1.4.3.4. Modifier ou remplacer les éléments d'une liste

Il est possible de modifier un élément particulier d'une liste en se servant de son indice. Par exemple considérons la liste définie comme suit :

```
x= ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
```

On souhaite ajouter 100 à l'élément 1800. Pour cela on fait :

```
x[3]=x[3]+100
print(x)  #  renvoie  ['lundi',  'mardi',  'mercredi',  1900,  20.357,
'jeudi', 'vendredi']
```

Pour remplacer « vendredi » par « vendredi saint », on fait :

```
x[6]=x[6]+ ' saint' # attention à l'espace dans ' saint' sinon on aura
vendredisaint.
print(x)  #  renvoie  ['lundi',  'mardi',  'mercredi',  1900,  20.357,
'jeudi', 'vendredi saint']
```

On peut carrément remplacer une valeur par une nouvelle. Exemple :

```
x[2]= 'dimanche' # remplace mercredi par dimanche
```

1.4.3.5. Ajouter de nouveaux éléments à une liste : fonction append()

Il est possible d'ajouter des nouveaux éléments en plus des éléments initiaux. Pour cela, on utilise la fonction append(). Exemple : Soit la liste initiale définie comme suit :

```
x= ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
```

On veut compléter cette liste en ajoutant samedi et dimanche. Pour cela on peut faire :

```
x.append('samedi')
x.append('dimanche')
```

Il faut juste signaler que les résultats de ces deux opérations pouvaient être obtenus en utilisant l'opérateur de concaténation + comme suit :

```
x=x+ ['samedi']
x=x+ ['dimanche ']
```

Ces deux opérations correspondent à l'addition de deux listes : la première liste constituée par la liste initiale x et la seconde liste formée d'un seul élément samedi ou dimanche. La liste finale ainsi obtenue est nommée x écrasant ainsi la liste x initiale.

Utilisation de la fonction extend()

Par ailleurs, comme on peut le constater la fonction append() ne peut ajouter qu'un seul élément à une liste à la fois. C'est pourquoi, on peut avoir recours à la fonction **extend()** lorsque l'on veut ajouter plusieurs éléments en même temps. Exemple :

```
x= ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
x.extend(['samedi', 'dimanche'])
print(x) # renvoie ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi',
'samedi', 'dimanche']
```

Avec la fonction extend(), les éléments à ajouter doivent être spécifiées sous forme de listes.

1.4.3.6. Insérer de nouveaux éléments dans une liste : fonction insert()

La fonction append() précédemment utilisée ajoute les nouveaux éléments à la fin de la liste initiale. En utilisant la fonction insert(), on peut insérer les nouveaux élément à n'importe quelle position dans la liste initiale en indiquant l'indice. Exemple : soit la liste initiale définie comme suit :

```
x= ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
```

On peut faire des ajouts suivants :

```
x.insert(2, 15) # Insère 15 dans la liste x à l'indice 2
print(x) # renvoie ['lundi', 'mardi', 15, 'mercredi', 1800, 20.357,
'jeudi', 'vendredi']
x.insert(7, 'samedi') # Insère samedi à l'indice 7 (dernière position
de la liste initiale)
print(x) # renvoie ['lundi', 'mardi', 'mercredi', 1800, 20.357,
'jeudi', 'vendredi', 'samedi']
```

1.4.3.7. Supprimer un élément d'une liste fonctions remove() ou delete

Il y a deux manières pour supprimer un élément d'une liste. La suppression en se basant sur la valeur de l'élément ou la suppression en se basant sur l'indice de l'élément. Dans le premier cas, on utilise la fonction remove() et dans le second cas, on utilise la fonction delete(). Les deux exemples ci-dessous sont des illustrations :

```
x= ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
x.remove('mercredi') # supprime 'mercredi' de la liste initiale x
del x.[2] # supprime l'élément d'indice 2 qui correspond ici à
'mercredi'
```

NB : Dans les deux exemples ci-dessus, même si la fonction remove et del permettent de supprimer le même élément, il faut noter que dans certains cas, les résultats renvoyés par les deux fonctions peut être radicalement différents. En effet, si l'élément de valeur 'mercredi' avait été dupliqué dans la liste initiale x, alors la fonction remove() aurait supprimé toutes les occurrences tandis que la fonction del n'aurait supprimer que l'occurrence située à l'indice 2. Il faut donc faire attention lors du choix entre les deux fonctions.

1.4.3.8. Renvoyer l'indice d'un élément en se basant sur sa valeur :fonction index()

Comme nous avons vu plus haut, les opérations d'indiçage (slicing) permettent de renvoyer les valeurs des éléments en indiquant leur indice. A présent, nous souhaitons renvoyer les indices en se basant sur les valeurs. La fonction index() permet donc de faire du contre-indiçage en renvoyant les indices des valeurs indiquées. Exemple :

```
x= ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
x.index('mercredi') # Renvoie 2 ( l'indice de l'élément mercredi)
x.index(20.357) # Renvoie 4 (l'indice de l'élément 20.357)
```

Nb : Lorsqu'il y a plusieurs occurrences de la même valeur dans la liste, la fonction index renvoie toujours l'indice de la première occurrence. On peut modifier ce comportement en se servant des options de la fonction index() que sont start et stop en y ajoutant d'autres fonctions.

1.4.3.9. Compter le nombre d'occurrences d'une valeur dans une liste : fonction count()

Il arrive fréquemment qu'une valeur se répète plusieurs fois dans une liste. Pour compter le nombre d'occurrences dans une liste, on utilise la fonction count(). Exemple :

```
x= ['lundi', 'mardi', 'lundi', 1800, 20.357, 'lundi', 'Mardi', 1800]
x.count('lundi') # Renvoie 3 ( le nombre d'occurrence de lundi)
x.count(20.357) # Renvoie 1
x.count(1800) # Renvoie 2
x.count('mardi') # Renvoie 1 (Attention : mardi et Mardi ne sont pas considérés comme les même valeurs)
```

1.4.3.10. Trier les éléments d'une liste : fonction sort()

Pour trier les éléments d'une liste dans un ordre croissant ou décroissant, on utilise la fonction sort(). Exemples : soit la liste x définie comme suit :

```
x=[ 12, 5 , 4 , 20 ,3, 8, 9, 51 ]
```

Pour trier x par ordre croissant, on fait :

```
x.sort() # renvoie [3, 4, 5, 8, 9, 12, 20, 51]
```

Et pour trier par ordre décroissant, on fait :

```
x.sort(reverse=True)  # Renvoie [51, 20, 12, 9, 8, 5, 4, 3]
```

Nb : Lorsque le tri est effectué sur une liste contenant uniquement les chaînes de caractères, le tri est réalisé par ordre alphabétique (croissant ou décroissant). Toutefois lorsque le tri est

réalisé sur une liste contenant à la fois des chiffres et des caractères, un message d'erreur est renvoyé. L'une des solution serait alors de convertir les chiffres en type caractères avec la fonction str() ; faire le tri et ensuite reconvertir les chiffres en utilisant les fonction int() ou float().

Trier les élements d'une liste en fonction des valeurs d'une autre liste

Exemple : Soit les deux listes suivantes :

```
X = ["a", "b", "c", "d", "e", "f", "g", "h", "i"]
Y = [ 0,   1,   1,    0,   1,   2,   2,   0,   1]
```

On souhaite trier la liste X selon les valeurs de la liste Y (par ordre croissant). On suit les étapes suivantes :

```
X_sorted=[x for (y,x) in sorted(zip(Y,X), key=lambda pair: pair[0], reverse=False)]
```

Cette fonction renvoie : ['a', 'd', 'h', 'b', 'c', 'e', 'i', 'f', 'g']

Pour effectuer un tri par ordre décroissant, on ajoute l'option reverse=True comme suit :

```
X_sorted=[x for (y,x) in sorted(zip(Y,X), key=lambda pair: pair[0], reverse=True)]
```

1.4.3.11. Intervertir l'ordre des éléments dans une liste : fonction reverse()

Intervertir l'ordre dans une liste c'est renverser l'ordre des éléments de sorte que le dernier élément de la liste initiale devient le premier élément de la liste finale et ainsi de suite. Pour réaliser cette opération, on utilise la fonction reverse(). Exemple :

```
x= ['lundi', 'mardi', 'lundi', 1800, 20.357, 'lundi', 'Mardi', 1800]
x.reverse() # Renvoie [1800, 'Mardi', 'lundi', 20.357, 1800, 'lundi', 'mardi', 'lundi']
```

1.4.3.12. Joindre les éléments d'une liste pour former une chaîne de caractères : fonction join()

La fonction join() permet d'associer tous les éléments d'une liste pour former une seule chaîne de caractères. La jointure des éléments peut se faire avec séparateur ou sans séparateur. Dans le cas de la jointure sans séparateur, tous les éléments sont concaténés les uns à la suite des autres sans aucun espace ni autre symbole. Exemple : Soit la liste x définie par :

```
x = ["A", "T", "G", "A", "T"]
```

On peut faire les différentes jointures suivantes :

Jointure avec espace comme séparateur

```
ch1= " ".join(x) # renvoie 'A T G A T'
```

Jointure avec tiret comme séparateur

```
ch2= "-".join(x) # Renvoie 'A-T-G-A-T'
```

Jointure sans séparateur (ni espace)

```
ch3= "".join(x) # Renvoie 'ATGAT'
```

Attention, la fonction join() ne s'applique qu'à une liste de chaînes de caractères (pas aux listes de chiffres). Pour ceux-ci il faut d'abord changer leur type en caractères en utilisant la fonction str().

1.4.3.13. Tester l'existence d'un élément dans une liste : fonction in

Pour vérifier si un élément défini par sa valeur se trouve dans une liste , on utilise la fonction in qui renvoie une valeur booléenne (True ou False) selon que l'élément existe ou pas dans la liste. Exemple : soit la liste définie par :

```
x= ['lundi', 'mardi', 'lundi', 1800, 20.357, 'lundi', 'Mardi', 1800]
```

On peut faire des tests d'existence suivants :

```
'lundi' in x # renvoie True (car il y a bien lundi dans la liste x)
'samedi' in x # renvoie False
1500 in x # renvoie False
1800 in x # renvoie True
200 not in x # renvoie True
```

La fonction « in » s'avère utile dans de nombreuses situations. Comme nous avons vu dans les sections précédentes, elle sert dans la définition des boucles « for » qui font des itérations sur tous éléments d'une séquence notamment d'une liste. Elle peut aussi servir dans la définition des instructions conditionnelles en l'associant avec la clause « if ». Exemple :

```
if 1500 in x :
     print("Oui 1500 existe dans la liste x")
else :
     print("Non 1500 ,n' existe pas dans la liste x")
```

1.4.3.14. Récupérer simultanément les indices et les valeurs des éléments d'une liste : fonction enumerate()

La fonction enumerate() permet de récupérer les indices et les valeurs d'une liste pour les stocker sous forme de tuple. Chacun des valeurs de ces deux variables peuvent être récupérées pour des utilisations futures.

Exemple 1 : Générer une liste à partir de la fonction enumerate()

Soit la liste x définie par :

```
x=['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']
```

On peut par exemple définir une nouvelle liste à partir des résultats de la fonction enumerate() comme suit :

```
mylist=list(enumerate(x,1))
print(mylist)  # renvoie [(1, 'Jean'), (2, 'Maximilien'), (3, 'Brigitte'), (4, 'Sonia'), (5, 'Jean-Pierre'), (6, 'Sandra')]
```

On constate alors que la fonction enumerate() crée toujours un couple de valeurs (indice/valeurs). Dans cet exemple, ces valeurs couples ont été récupérées pour former une liste nommée mylist.
Nb : L'option 1 dans le fonction enumerate() signifie que les indices nouvellement crées doivent commencer par 1 au lieu de la valeur par défaut qui est 0.

Exemple 2 : Construire une boucle « for » à partir de la fonction enumerate()

Soit la liste x définie précédemment, on peut élaborer des instructions en boucle avec la fonction enumerate() comme suit :

```
for i, v in enumerate(x):
      print(i)
      print(v)
```

Dans cet exemple, on récupère chaque indice et valeur de la liste x et on les affiche en utilisant la fonction print(). On pouvait aussi définir d'autres instructions en boucle. Par exemple définir deux nouvelles listes : une constituée par les indices de la liste initiale et l'autre constituée par les valeurs. Voir exemple ci-dessous :

```
x=['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']

listIndices=[] # Création d'une liste vide (pour les indices)
listValeurs=[] # Création d'une liste vide (pour les valeurs)
for i, v in enumerate(x):
      listIndices.append(i)
      listValeurs.append(v)
print(listIndices)
print(listValeurs)
```

Remarque : la fonction append() utilisée dans la boucle for constitue une manière originale pour faire la copie d'une liste initiale qui paraît compliquer dans certaines situations (voir section sur la copie de liste un peu plus bas).

1.4.3.15. Faire la copie d'une liste

L'une des particularités d'une liste est que l'opération d'assignation (=) ne permet pas faire la copie d'une liste. Il permet seulement de référencer les valeurs de celle-ci. Par exemple, soit x une liste définie initialement. En définissant une liste y telle que y=x crée en réalité une référence et non une copie car en modifiant un élément de x, cela se répercute directement sur y sans aucun besoin de modification supplémentaire. Pour s'en rendre compte, penons les exemples suivants :

```
x = [1,2,3] # définition de la liste x
y = x # définition de la liste y par assignation
print(y) # renvoie [1,2,3]
x[1] = -15 # Modification de la liste x ( le second élément remplacé par -15)
print(y) # renvoie [1,-15,3].
```

On voit que cette modification de x se répercute automatiquement sur y. Ce qui démontre que la création de copie par assignation ne permet de faire qu'une copie dynamique (référencement) et non une copie définitive.

Pour faire la copy d'une liste, on peut se servir de la fonction copy() comme suit :

Il existe néanmoins plusieurs méthodes alternatives pour faire une copie d'une liste. Le choix de la méthode dépendra alors selon qu'il s'agisse d'une liste à une dimension ou une liste à plusieurs dimensions (c'est-à-dire une liste des listes).

Copie d'une liste à une dimension

La première méthode pour faire la copy d'une liste simple est d'utiliser la fonction copy(). Celle-ci est illustrée par l'exemple 1 ci-dessous

Exemple 1 : Utilisation de la fonction copy()

```
x = [1,2,3] # définition de la liste x
y = x.copy() # définition de la liste y en utilisant la fonction copy
print(y) # renvoie [1,2,3]
x[1] = -15 # Modification de la liste x ( le second élément remplacé par -15)
print(y) # renvoie [1,2,3].
```

Attention : La fonction copy() ne fait qu'une copie superficielle de la liste. En ce sens, elle ne fonctionne convenablement que lorsqu'il s'agit d'une liste simple (liste à une dimension). Elle n'est donc pas valable pour une liste à plusieurs dimensions.

Il existe néanmoins plusieurs autres méthodes plus ou moins conventionnelles pour faire une copie rapide d'une liste à une dimension. Les exemples suivants sont des illustrations.

Exemple 2 : Utilisation de la fonction list()

```
x = [1,2,3] # définition de la liste initiale x
```

```
y = list(x) # définition de la liste y en utilisant la fonction list() sur x
x[1] = -15 # Modification de la liste x
print(y) # renvoie [1,2,3].
```

On remarque qu'en utilisant la fonction list(), toute modification effectuée ultérieurement sur x ne se répercute plus sur y. Il faut toutefois noter que l'utilisation de la fonction list() ne fonctionne que dans le cas d'une liste à une dimension (pas dans le cas d'une liste à plusieurs dimensions).

Exemple 3 : Utilisation de l'opérateur de slicing [:]

Pour sélectionner tous les éléments de x et les stocker dans une nouvelle liste assigner à un nouveau nom, il suffit d'utiliser l'opérateur [:] comme suit :

```
x = [1,2,3] # définition de la liste initiale x
y = x[:] # définition de la liste y en sélectionnant tous les élements de x
x[1] = -15 # Modification de la liste x
print(y) # renvoie [1,2,3].
```

Là encore, les modifications de x ne se répercutent plus sur y. Toutefois cette méthode reste applicable uniquement dans le cas d'une liste à une dimension.

Exemple 4 : Utilisation de la boucle « for » et de la fonction append()

```
x = [1,2,3] # définition de la liste initiale x
y = [] # Création d'une liste vide
for i in range(len(x)):
      y.append(x[i])
print(y)
```

Cette méthode reste aussi valable uniquement dans le cas d'une liste à une dimension.

Exemple 5 : Utilisation de la fonction deepcopy()

deepcopy() est une fonction avancée appartenant au module copy qui permet de faire les copies des objets python. On peut donc utiliser cette fonction pour la copie d'une liste. Toutefois, pour l'utiliser, il faut d'abord importer le module copy comme suit :

```
import copy
x = [1,2,3] # définition de la liste initiale x
y = copy.deepcopy(x) # définition de la liste y en utilisant deepcopy()
print(y) # renvoie [1,2,3].
```

Copie d'une liste à plusieurs dimensions

Pour faire la copie d'une liste à plusieurs dimensions, on utilise la fonction deepcopy() du module copy. Cette fonction est valable quelle que soit la dimension de la liste. L'exemple ci-dessous illustre la création d'une copie d'une liste à deux dimensions.

```
import copy
x = [[1,2],[3,4]] # définition de la liste initiale x
y = copy.deepcopy(x) # définition de la liste y en utilisant
deepcopy()
print(y) # renvoie [[[1,2],[3,4]]

x[1][1] = -15 # Modification de la liste x
print(y) # renvoie [[[1,2],[3,4]]
```

NB : Il existe plusieurs autres fonctions applicables sur les listes. Pour accéder à la liste complètes des fonctions associées à une liste, il faut utiliser la fonction dir(). Par exemple la fonction dir(x) affiche toutes les fonctions disponibles pour la liste x. Et pour obtenir l'aide sur une fonction on tape la commande help(x.nomFonction). Exemple :

```
help(x.sort) # affiche l'aide la fonction sort() applicable à la liste
x.
```

Attention : la plupart des fonctions qui ont été présentées ci-dessus (remove, sort, reverse, etc…) font des modifications de type « IN PLACE » sur la liste indiquée. Ce qui signifie que toute application de la fonction remplace la liste initiale par la liste modifiée. Cette situation ne convient pas dans tous les cas. C'est pourquoi, il est préférable, si nécessaire, de créer d'abord une copie de la liste la liste initiale et ensuite réalisée la modification sur cette copie créée.

1.4.3.16. Combiner les éléments de deux listes pour former une liste de couple d'éléments : fonction zip()

Nous avons vu précédemment que la fonction enumerate() renvoyait les indices et les valeurs d'une liste pour former un couple de valeur indice/ valeur organisé sous forme de liste. La fonction zip() permet de combiner les éléments de deux listes en formant une liste de couple formés par les éléments de même indice. Très concrètement, considérons les deux listes suivantes :

```
x1=['a', 'b', 'c']
x2=['d', 'e', 'f']
```

En appliquant la fonction zip() sur ces deux liste telle que :

```
y=list(zip(x1, x2))
```

On obtient une liste qui se présente telle que : [('a', 'd'), ('b', 'e'), ('c', 'f')]

Le résultat renvoyé par zip() est une liste de tuples tout comme pour enumerate()

NB : Puisque la fonction zip() forment les couples de valeurs à partir des éléments de même indice, lorsque les deux liste ne sont pas de même longueur, la dimension de la liste finale se limite à celle de la liste avec la longueur minimale. Les autres éléments de la grande liste sont donc ignorés.

Par ailleurs, les résultats renvoyés par la fonction zip() sont souvent utilisés pour exécuter d'autres instructions comme pour la fonction enumerate(). La structure générale de ces instructions se présente alors comme suit :

```
for e1, e2 in zip(liste1, liste2):
    instructions utilisant e1 et e2
```

Notons aussi que les valeurs réalisées par la fonction zip() peuvent être explicitement formulées comme suit :

```
for i in range(min(len(liste1), len(liste2))):
     e1, e2 = liste1[i], liste2[i]
     instructions utilisant e1 et e2
```

Dans cette formulation, on définit une boucle allant de 0 à la valeur minimum entre len(liste1) et len(list2). Ensuite, on récupère les valeurs de liste1 et liste2 correspondant à l'index i. Ces valeurs sont alors utilisées pour définir les instructions. Une telle formulation montre donc que l'utilisation de la fonction zip() n'est pas un passage obligé.

la fonction unzip()

Il arrive souvent que l'on veuille décomposer une liste de tuple (liste zippée) en deux listes distinctes. Il faut alors utiliser la méthode unzip. Malheureusement, il n'y a pas sous python une fonction directe qui puissent réaliser cette tâche, il faut utiliser une combinaison de fonctions y compris la fonction zip() elle-même. L'exemple ci-dessous est une illustration.

1- On définit d'abord une fonction lambda nommée myUnzip

```
myUnzip = lambda x: [list(i) for i in zip(*x)]
```

2- Ensuite, on applique cette fonction sur une liste zippée. Ex soit la liste zipListe définie comme suit :

```
zipList=[(1, 2), (3, 4), (5, 6)]
```

En appliquant la fonction myUnzip sur cette liste, on obtient une nouvelle liste nommée unzipList telle que :

```
unzipList= myUnzip(zipList) # Renvoie [[1, 3, 5], [2, 4, 6]]
```

3- Comme unzipList est une liste à deux dimensions constituée de deux éléments, on fait du slicing sur cette liste pour former deux listes séparées. On a :

```
liste1=unzipList[0]
liste2=unzipList[1]
print(liste1) # renvoie [1, 3, 5]
print(liste2) #renvoie  [2, 4, 6]
```

Penser donc à utiliser cette méthode chaque fois qu'on dispose d'une liste de tuple et qu'on souhaite la décomposer en deux listes distinctes.

1.5. Les fonctions map(), filter() et reduce() pour le traitement des listes

Dans cette section, nous présentons quelques fonctions qui peuvent s'avérer utiles lors du traitement des éléments d'une liste. Il s'agit notamment des fonctions map(), filter() et reduce().

1.5.1. La fonction map()

La fonction map() permet d'exécuter une fonction donnée sur chaque élément d'une liste initiale et renvoie les résultats finals sous forme d'une nouvelle liste. La fonction map() fonctionne sur le principe définit par le programme suivant :

```
def map(f, inputList):
      outputList = []
      for x in inputList:
            outputList.append(f(x))
      return outputList
```

Dans cette architecture, on définit une fonction qui s'appelle map(). Nous reviendrons plus en détail sur la définition des fonctions. La fonction map() prend deux arguments : une fonction f et une liste initiale inputList. Le but est d'appliquer la fonction f sur chaque élément de la liste inputList en utilisant une boucle for. Ensuite stocker au fur et à mesure le résultat obtenu dans une liste initialement vide outputList. Et enfin, renvoyer la liste finale obtenue lorsque la boucle sera bouclée. C'est le principe de la fonction map().

Le résultat final renvoyé par la fonction map() dépendra alors du choix de la fonction f. Plusieurs cas peuvent se présenter ; par exemple compter la longueur de chaque élément, compter le nombre d'occurrence de chaque élément, etc...Il faut noter que la fonction map() est beaucoup utilisée dans le traitement des données de masse. Par exemple, l'essentiel de l'algorithme Hadoop MapReduce écrit sous java est basée sur ce principe.

Les exemples ci-dessous illustrent quelques utilisation simples de la fonction map() sous python.

Exemple 1 : Déterminer la longueur de chaque mot dans une liste de mots

Soit une liste l définie par:

```
l = ["le", "kangourou", "vert"]
```

Nous souhaitons créer une liste contenant les longueurs des éléments. Pour cela, nous allons utiliser la fonction len en spécifiant la liste comme suit :

```
x=list(map(len, l))
print(x) # renvoie [2, 9, 4]
```

Dans cet exemple, la fonction list() sert uniquement à récupérer les résultats renvoyés par la fonction map() sous forme de liste

Exemple 2 : Convertir tous les éléments d'une liste de chaine de caractères en minuscule ou en majuscule

```
l = ["Le", "Kangourou", "Vert"]
x=list(map(str.lower, l)) # convertir en minuscule
y=list(map(str.upper, l)) # convertir en majuscule
print(x) # renvoie ['le', 'kangourou', 'vert']
print(y) # renvoie ['LE', 'KANGOUROU', 'VERT']
```

Exemple 3: Calculer la racine carrée des éléments d'une liste

```
l= [1,4,9,16] # définition de la liste initiale
from math import sqrt # importation de la fonction sqrt à partir du
module math
x=list(map(sqrt, l)) # renvoie [1.0,2.0,3.0,4.0]
```

Nb : on peut aussi utiliser d'autres opérations, log, exp, etc...

Exemple 4 : Application d'une fonction définie par l'utilisateur

Dans les exemple ci-dessus, la fonction map() a été exécutée à partir des fonctions déjà disponibles dans les modules standards de python. Mais dans de nombreuses situations, l'utilisateur est amené à écrire sa propre fonction pour pouvoir définir les tâches qu'il souhaite appliquer avec la fonction map(). L'exemple ci-dessus illustre l'écriture et l'utilisation des fonctions de l'utilisateur (nous reviendrons plus tard sur la définition des fonctions).

On souhaite appliquer la fonction map() sur une fonction qui renvoie le reste de la division par 2 des éléments d'une liste définie par l=[2, 5, 7, 17, 33, 21, 12, 50].

Dans ce cas, nous allons d'abord définir une fonction qui renvoie le reste de la division par 2. On suit les étapes suivantes :

```
def restFunction(x) :
     return x % 2
```

Nous avons nommé la fonction restFunction, qui prend comme argument x et qui renvoie le reste de la division de x par 2 avec l'opérateur modulo (%). En soumettant cette fonction à la fonction map(), x prendra successivement les valeurs de la liste initiale l. Ainsi on a :

```
restList = list(map (restFunction, l))
print(restList) # renvoie [0, 1, 1, 1, 1, 1, 0, 0]
```

1.5.2. La fonction filter()

Comme sa dénomination indique, la fonction filter() permet de mettre un filtre sur les éléments d'une liste afin de renvoyer ceux qui remplissent les condition définie par le filtre. Contrairement à la fonction map(), la fonction filter() exécute toujours des instructions conditionnelles. Les exemples ci-dessous sont des illustrations.

Exemple 1 : Renvoyer tous éléments alphabétiques dans une liste

Soit la liste l définie par :

```
l=['x', 'y', '2', '3', 'a'] # On remarque que tous les élements sont en chaînes de caractères (mais ne sont pas tous alphabétiques)
```

On veut construire une nouvelle liste contenant uniquement les éléments alphabétiques (excluant alors les éléments numériques). On fait :

```
x=list(filter(str.isalpha,l ))
print(x) # renvoie ['x', 'y', 'a']
```

La fonction isalpha est un opérateur booléen qui vérifie si une chaine de caractères donnée est alphabétique ou non (nous reviendrons plus en détails sur les chaines de caractères dans les sections suivantes).

Exemple 2 : Récupérer les nombres impairs d'une liste

Soit la liste initiale définie par :

l=[2, 5, 7, 17, 33, 21, 12, 50]

On souhaite appliquer la fonction filter() à partir d'une fonction qui identifie les nombres impairs dans cette liste. Nous allons ici devoir élaborer une petite fonction qui effectue cette tâche. On a :

```
def impair(x):
    if x % 2 !=0:
        return x
impairList=list(filter(impair, l))
print(impairList) # renvoie [5, 7, 17, 33, 21]
```

Exemple 3 : Renvoyer tous les éléments d'une liste qui contiennent un motif donné.

Soit la liste définie par :

```
l = [ "le", "kangourou", "vert"]
```

On souhaite récupérer tous les éléments de la liste qui contiennent "e".

Pour cela, on définit d'abord une petite fonction comme suit :

```
f= lambda x: "e" in x
eList=list(filter(f, l))
print(eList) # renvoie ['le', 'vert']
```

Dans cet exemple, remarquer l'utilisation de la fonction lambda. En effet, comme il s'agit ici d'une instruction simple, il n'est pas nécessaire d'écrire une fonction classique avec le mot clé def. On peut se contenter de la fonction lambda qui est une fonction anonyme (sans nom défini préalable) et qui renvoie toujours un résultat équivalent à ce que fait la fonction return dans une fonction traditionnelle. Dans notre cas ici, nous avons décidé d'attribuer le nom f à la fonction lambda afin de simplifier la spécification de la fonction filter(). D'une manière générale, la structure d'une fonction lambda est toujours conçu pour renvoyer une valeur x lorsqu'une condition y est vérifiée. Dans notre cas ici, on indique de renvoyer x lorsque x contient e. En appliquant cette instruction sur la liste l, on obtient alors ['le', 'vert'] qui sont les deux éléments qui contiennent "e".

Exemple 4 : Renvoyer tous les éléments d'une liste dont la longueur est supérieure à une certaines valeurs.

Soit la liste l suivante :

```
l = [ "le", "kangourou", "vert"]
```

On souhaite renvoyer tous les éléments dont la longueur est supérieure à 5.

Ici aussi, on peut utiliser la fonction lambda puisque l'instruction à exécuter ne nécessite forcement d'élaborer une fonction traditionnelle. Ainsi, on a :

```
f= lambda x: len(x)>5
lenList=list(filter(f, l))
print(lenList) # renvoie ['kangourou']
```

1.5.3. La fonction reduce()

La fonction reduce() est la troisième fonction du trio map()-filter()-reduce() dont le rôle est de renvoyer une information synthétique à partir d'un ensemble de valeurs définie par une liste.

D'une manière générale, la fonction reduce() renvoie généralement une valeur tandis que les fonction map() et filter() renvoient des listes.

Il faut noter que dans les version récentes de python la fonction reduce() n'est plus une fonction de la bibliothèque standard de python. La fonction a été intégrée au module functools qui nécessite d'abord d'être importé avant de pouvoir utiliser la fonction reduce(). A cet effet, nous utilisons d'abord la fonction import telle que :

```
import functools
```

```
from functools import reduce
```

Les exemples ci-dessus illustrent quelques utilisation de la fonction reduce()

Exemple 1 : Les opérations mathématiques courantes :

Maximum d'une liste

```
reduce(max, [5,8,3,1]) # renvoie 8
```

Somme des éléments d'une liste

```
import operator
```

```
reduce(operator.add, [1,2,3]) # renvoie 6
```

Pour voir la liste complète des opérateurs arithmétiques de python sous forme de fonctions consulter la page : https://docs.python.org/2/library/operator.html

Exemple 2 : Effectuer des opérations sur les listes

Concaténation de listes

```
reduce(operator.concat, [[1,2],[3,4],[],[5]], []) # renvoie [1,2,3,4,5]
```

Jointure des éléments d'une liste

```
f=lambda s,x: s+str(x)
reduce(f, [1,2,3,4], '') # renvoie '1234'
```

1.6. Etude des objets « tuples »

A l'instar des objets listes, les objets tuples sont des séquences de valeurs (numériques et/ou en caractères) dont les éléments sont indiqués entre parenthèses (crochets pour les listes) et séparés par des virgules. A la différence des listes, les éléments d'un tuple sont indicés (accessibles par slicing) mais pas modifiables. Toutefois, il est possible d'extraire les éléments pour en créer d'autres objets.

1.6.1. Définition d'un tuple

Les exemples ci-dessous illustrent quelques définitions de tuples.

```
x = (1, 2, 3, 4, 12, 5, 7, 31) # Crée un tuple de valeurs numériques
y=('lundi','mardi','mercredi','jeudi','vendredi') # tuple de chaines de caractères
z=('lundi','mardi','mercredi',1800,20.357,'jeudi','vendredi')  #  tuple de valeurs numériques et de caractères
k=tuple(range(4,13)) # tuple de valeur numériques entre 4 et 12
t=tuple()
```

Ces cinq exemples montrent différentes manières de définition d'un tuple. La variable x est un tuple constitué uniquement de valeurs numériques. La variable y est constituée uniquement de valeurs en caractères. La variable z est un tuple qui combine les valeurs numériques et en caractères.

S'agissant de la variable k, c'est un tuple crée en utilisant la fonction tuple() à l'image de la fonction list(). Ici, le tuple est formé par une séquence de valeurs numérique générée par la fonction range().

Quant à la variable t, c'est un tuple vide qui s'obtient en utilisant la fonction tuple() sans argument.

Remarque : Pour créer un tuple d'un seul élément, il faut utiliser une syntaxe avec une virgule définie sous la (valeur_element,). Cela permet d'éviter l'ambiguïté avec la définition d'une simple variable.

```
y = (10,) # Crée un tuple constitué d'un seul élement 10
y=(10) # crée une variable simple dont la valeur est 10
```

Noter aussi qu'il est possible d'en créer un tuple sans les parenthèses, dès lors que ceci ne pose pas d'ambiguïté avec une autre expression telle que la création de variable avec assignation parallèle. Exemple :

```
x = 1,2,3 # Crée un tuple dont les éléments sont 1, 2 et 3
x, y, z=1,2,3 # Crée trois variables x, y, et z dont les valeurs sont respectivement 1, 2 et 3
```

1.6.2. Indiçage des tuples (slicing)

Comme signalé précédemment, les éléments d'un tuple sont indicés mais ils ne sont pas modifiables. Néanmoins, il est possible d'accéder aux éléments, les extraire pour d'autres utilisations (créations de variables simples, de listes, ou de nouvelles tuples, etc..).

Le slicing d'un tuple se fait de la même manière que dans une liste. Les exemples ci-dessous sont des illustrations.

Soit le tuple x défini comme suit :

```
x=('lundi','mardi','mercredi',1800,20.357,'jeudi','vendredi')
print(x) # affiche tous les éléments du tuple x
x[0] # renvoie le premier élément de x :lundi (Nb :l'indiçage commence toujours à 0)
x[3] # renvoie l'élément d'indice 3( quatrième élément de x) :1800
x[1:3] # Renvoie tous éléments compris entre l'indice 1 et l'indice 3 (Nb : l'élément d'indice 3 est exclu)
x[1:6 :2] # Renvoie tous éléments compris entre l'indice 1 et l'indice 6 avec un saut de 2 éléments à chaque fois ['mardi', 1800, 'jeudi'] (l'élément d'indice 6 est exclu).

x[2 :] # renvoie tous éléments à partir de l'élément d'indice 2 ( inclu).
x[:3] # renvoie tous éléments situés avant l'élément d'indice 3 (exclu)
x[-1] # Indiçage négatif, renvoie le dernier élément du tuple (équivaut ici à x[6])
x[-2] # Indiçage négatif, renvoie l'avant-dernier élément du tuple (équivaut ici à x[5])
```

Nb : Les éléments renvoyés par slicing peuvent être stockés dans de nouvelles variables, des listes ou dans d'autres tuples.

1.7. Etude des objets « set »

Les objets set sont des séquences de valeurs sans doublons (numériques et/ou en caractères) indiquées entre accolades et séparés par des virgules. A la différence des listes et des tuples, les set ne sont pas indicés. Les éléments sont inaccessibles par slicing. Par conséquent, ils sont non modifiables. Pour pouvoir effectuer les opérations de modifications, on est souvent obligé de transformer le set en liste en utilisant la fonction list(). Nous reviendrons sur ces aspects plus bas.

D'une manière générale, les sets sont construits à partir des listes ou des tuples. Ils sont principalement utilisés soit pour éliminer des doublons dans une séquence, soit pour tester si un élément appartient ou non à une collection. Contrairement aux autres objets, les set permettent de réaliser des opérations mathématiques sur les ensembles (union, intersection, la différence....).

Les exemples ci-dessous illustrent quelques définitions de set.

```
x = {1, 2, 3, 2, 4,7, 12, 5, 7, 31, 7} # Crée un set de valeurs numériques en supprimant les doublons de 2 et de 7
y={'lundi','mardi','mercredi','jeudi','vendredi','mardi'} # set de chaines de caractères en supprimant le doublon mardi
myList=('lundi','mardi','mercredi',1800,20.357,'jeudi',1800,
'vendredi','vendredi') # définition d'une liste
z=set(myList) # Crée un set à partir d'une liste
k=set() # Crée un set vide
```

Ces exemples montrent différentes manières de définition d'un set. La variable x est un set constitué uniquement de valeurs numériques (la séquence de valeurs renvoyée ne contiendra aucun doublon). La variable y est constituée uniquement de valeurs en caractères (sans doublon). La variable z est un set crée à partir d'une liste en utilisant la fonction set(). Et la variable k est un tuple vide

Remarque : Etant donné que les éléments d'un set sont inaccessibles et non modifiables, pour pouvoir appliquer les opérations de modification, il faut d'abord transformer le set en liste, faire les modifications et reconvertir la liste modifiée en set si utile.

Par ailleurs, même si on ne peut pas modifier les éléments d'un set, on peut effectuer les opérations sur les ensembles (union, intersection, etc…) Voici-ci-dessous quelques exemples.

Soit x et y deux set définis comme suit :

```
x=set('abracadabra') # un set formé par les éléments de la chaine, renvoie {'c', 'a', 'b', 'r', 'd'}
y=set('alacazam') # un set formé par les éléments de la chaine, renvoie {'c', 'z', 'a', 'l', 'm'}
print(x-y) # lettre dans x mais pas dans y, renvoie {'b', 'r', 'd'}
print(x|y) # lettres dans x ou dans y, renvoie {'b', 'r', 'm', 'd', 'z', 'l', 'c', 'a'}
print(x & y) # lettre dans x et dans y, renvoie {'c', 'a'}
print(x^y) # lettre qui uniquement dans x ou uniquement dans y (mais pas dans les deux en même temps), renvoie {'b', 'r', 'm', 'd', 'z', 'l'}
```

1.8. Etude des objets « array »

Les objets « array » sont des séquences de valeurs (uniquement numériques) se présentant sous formes de tableaux d'un ou plusieurs dimensions (vecteur ou matrice). Les arrays sont objets définis avec la fonction array() accessible à partir du module numpy (module spécialisé dans les calculs numériques sous python). Pour pouvoir donc définir un objet array, il faut, au préalable importer le module numpy comme suit :

```
import numpy
```

1.8.1. Définition d'un array

Les objets array sont généralement définis à partir des objets listes, tuples ou tout autre séquence de valeurs comme les range. Ceux-ci doivent être spécifiées comme argument de la fonction array().

On peut distinguer les arrays selon leur dimension : les arrays à une dimension, les arrays à deux ou plusieurs dimensions.

1.8.1.1. Définition d'un array à 1 dimension

Les exemples ci-dessous illustrent quelques cas de définition d'un array à une dimension.

```
import numpy # Importe le module numpy pour pouvoir utiliser la fonction array()
myList= [1,2,3, 8, 24, 12] # Crée une liste nommée myList
myTuple= (0,2,3, 9, 7, 13) # Crée un tuple nommée myTuple
x= numpy.array(myList) # Crée un array à partir d'une liste, renvoie : array([ 1,  2,  3,  8, 24, 12])
y= numpy.array(myTuple) # Crée un array à partir d'un tuple, renvoie : array([ 0,  2,  3,  9,  7, 13])
z=numpy.arange(10) # Crée un array en utilisant la fonction arange(). Renvoie : array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

La fonction arange() est l'équivalent de la fonction range() pour les cas des arrays. Elle permet directement de générer un array à partir de la séquence de valeurs indiquée. Par conséquent l'expression z=numpy.arange(10) est un raccourci de l'expression suivante :

```
z=numpy.array(list(range(10))) # Renvoie : array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Toutefois, l'avantage de la fonction arange() par rapport à la fonction range(), c'est que celle-ci peut prendre des arguments numériques de type float ou interger alors que la fonction range() ne prend que des integers. Exemple :

```
z=numpy.arange(2.5,4.5, 0.5) # Array de séquence numérique compris entre 2.5 et 4.5 avec un pas de 0.5 Renvoie : array([ 2.5,  3. ,  3.5,  4. ])
```

La différence fondamentale entre un objet array à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un vecteur. Par conséquent on peut effectuer des opérations vectorielle élément par élément. Ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Exemples :

Soit le array x défini par :

```
x = numpy.arange(4) # renvoie array([0, 1, 2, 3])
x + 1 # renvoie array([1, 2, 3, 4])
x + 0.1 # renvoie array([ 0.1, 1.1, 2.1, 3.1])
x * 2 # renvoie  array([0, 2, 4, 6])
x * x # renvoie array([0, 1, 4, 9])
```

Sur le dernier exemple de multiplication, l'array final correspond à la multiplication élément par élément des deux array initiaux. Avec les listes, ces opérations n'auraient été possibles qu'en utilisant des boucles.

1.8.1.2. Définition d'un array à plusieurs dimensions

Comme pour les listes, un array à deux dimensions se définit simplement comme array de array c'est-à-dire un array dont les éléments individuels sont constitués par des arrays. Les exemples ci-dessous illustrent quelques cas de définition d'un array à plusieurs dimensions.

Array à deux dimensions :

Pour créer un array à deux dimensions à partir d'une liste, il suffit de passer en argument une liste de listes à la fonction array(). Exemple :

```
myList=[[1,2,3],[2,3,4],[3,4,5]] # définition d'une liste de listes
x=numpy.array(myList) # Définition d'un array d'array.
```

Nb: un array à deux dimensions ne veut pas dire array à deux colonnes. Il suffit simplement un array constitué de liste de liste.

Array à trois dimensions :

On peut aussi créer des tableaux à trois dimensions en passant à la fonction array() une liste de listes de listes :

```
myList=[[[1,2],[2,3]],[[4,5],[5,6]]]  #  définition  d'une  liste  de
listes
x=numpy.array(myList) # Définition d'un array d'array d'array (array à
3 dimensions).
```

1.8.1.3. Indiçage d'un array (slicing)

L'indiçage d'un array se fait de la même manière que dans le cas d'une liste. Les exemples ci-dessous sont des illustrations.

Cas d'un array à une dimension

```
x = numpy.arange(10) # Définition d'un array, renvoie array([0, 1, 2,
3, 4, 5, 6, 7, 8, 9])
x[1] # renvoie l'élément d'indice 1 (correspond ici à 1)
```

```
x[5:]  # renvoie tous les éléments à partir de l'élément d'indice 5, ici array([5, 6, 7, 8, 9])
 x[: : 2] # renvoie tous les éléments avec un saut de 2 élements à chaque fois array([0, 2, 4, 6, 8])
```

Cas d'un array à deux dimensions

```
y = numpy.array([[1,2],[3,4]]) # définition d'un array à deux dimensions
y[0,:] # Sélectionne tous les éléments de l'array qui a l'indice 0 dans les arrays de second niveau, renvoie array([1, 2])
y[:,0] # sélectionne tous les éléments se trouvant à l'indice 0 dans les array de second niveau, renvoie array([1, 3])
y[1,1] # Sélectionne l'élément d'indice 1 de l'array qui a l'indice 1 dans la liste des array de second niveau, renvoie  4
```

1.8.2. Déterminer la dimension d'array

Pour déterminer la dimension d'un array, on utilise la fonction shape(). Voir exemple ci-dessous :

Soit le array x défini par :

```
x = numpy.arange(3)
numpy.shape(x) # renvoie (3,) signifiant que l'array x contient trois élements
```

1.8.3. Les opérations sur les arrays

Cette section décrits quelques opérations couramment effectuées sur les arrays.

1.8.3.1. Conversion d'un array vectoriel en array matriciel: fonction reshape() et resize()

les fonctions reshape() et resize()permettent de transformer un array qui se présente initialement comme un vecteur en un array matriciel en indiquant le nombre de lignes et de colonnes. Les exemples ci-dessous illustrent le type de redimensionnement qu'effectue chacune de ces deux fonctions.

```
x = numpy.arange(9) # définit un array de 9 élements allant de 0 à 8
numpy.reshape(x,(3,3)) # Renvoie un array matriciel 3x3 en distribuant les éléments de l'array initial sur chaque ligne en revenant à la ligne quand la ligne est remplie.
numpy.resize(x,(2,2)) # Renvoie un array matriciel 2x2 en distribuant les éléments de l'array initial sur chaque ligne en revenant à la ligne quand la ligne est remplie. Il s'arrête lorsque la matrice 2x2 est remplie.
```

```
numpy.resize(x,(4,4)) # Renvoie un array matriciel 4x4 en distribuant les éléments de l’array initial sur chaque ligne en revenant à la ligne quand la ligne est remplie. Il continue le remplissage tant que la matrice 4x4  n’est pas totalement remplie. Le remplissage continue à partir de la première valeur de l’array initial.
```

Nb : La fonction reshape() nécessite que la dimension de la matrice indiquée soit compatible avec le nombre d’éléments contenus dans l’array de départ. Ce qui n’est pas obligatoire pour la fonction que resize() qui puise les valeurs en recommençant à partir du premier élément de l’array initial. Et lorsque la dimension de la matrice est plus petite que la dimension de l’array de départ, le remplissage avec la fonction resize() s’arrête lorsque le remplissage de la matrice est complet. Pour le cas de la fonction reshape, celle-ci renvoie une erreur.

1.8.3.2. Transposé d'un array matriciel: fonction transpose()

la fonction transpose() renvoie la transposée d’un array. Par exemple pour une matrice :

```
x = numpy.arange(9) # génère un array
y=numpy.reshape(x,(3,3)) # tranforme l’array en un array matriciel 3x3
z=numpy.transpose(y) # Effectue la transposée de la matrice y
```

1.8.3.3. Création d'un array matriciel rempli de 0 ou 1: fonctions zeros() et ones()

Les fonctions zeros() et ones() permettent de construire des objets array contenant des 0 ou de 1, respectivement. Pour cela, il faut leur passer un tuple indiquant la dimensionnalité voulue. Les exemples ci-dessus sont illustrations.

```
numpy.zeros((3,3)) # Crée un array matriciel de dimension 3x3 rempli de 0 (de type float)
numpy.zeros((3,3),int) # Crée un array matriciel de dimension 3x3 rempli de 0 (de type integer)
numpy.ones((3,3)) # Crée un array matriciel de dimension 3x3 rempli de 0 (de type float)
numpy.ones((3,3), int) # Crée un array matriciel de dimension 3x3 rempli de 0 (de type float)
```

Nb : Par défaut, les fonctions zeros() et ones() génèrent des réels (float), mais on peut modifier ce comportement en demandant par exemple des entiers en passant l’option int en second argument.

1.8.3.4. Les opérations algébriques (matricielles) sur les arrays

Les arrays étant par définitions des objets vectoriels ou matriciels, on peut effectuer des opérations algébriques comme le produit matriciel, le calcul de déterminant, l'inversion de matrice, etc… Dans la plupart de ces opérations, on utilisera la fonction linalg du module numpy. Les exemples ci-dessous sont des illustrations.

Soit l'array matriciel x défini comme suit :

```
x = numpy.resize(numpy.arange(4),(2,2)) # génère un array matriciel
```

On peut effectuer les opérations algébriques suivantes :

```
x2=numpy.dot(a,a) # renvoie le produit matriciel entre x et x alors
x*x aurait renvoyé le produit élément par élément.
numpy.linalg.inv(x) # renvoie  l'inverse de la matrice x
numpy.linalg.det(x) # renvoie le déterminant de la matrice x
numpy.linalg.eig(x)  # renvoie un tuple dont le premier élément
correspond aux valeurs propres et le second élément aux vecteurs
propres.
```

Pour avoir plus de détails sur la fonction linalg, consulter la page suivante : http://docs.scipy.org/doc/numpy/reference/routines.linalg.html

1.8.3.5. Utilisation de la fonction argsort() sur un array

La fonction argsort() renvoie sous forme de liste (ou d'array) les indices qui auraient trier u l'array qui a été fournit en arguement. Exemple :

```
x = numpy.array([1.48,1.41,0.0,0.1]) # Crée un array
print (x.argsort()) # renvoie [2 3 1 0]
```

[2, 3, 1, 0] indique que l'élément le plus petit est à l'index 2, le suivant le plus petit à l'index 3, puis l'indice 1, puis le plus grand est à l'indice 0. La fonction renvoie donc les indices plutôt que les valeurs.

1.9. Etude des objets « dictionnaire »

Dans une conception pythonienne, un dictionnaire (encore appelé bibilothèque) est une collection d'objets (généralement des listes, des tuples, des array) définie sur la base clé-valeur. Contrairement aux listes, les tuples ou les array dans lesquels les éléments sont identifiés à partir de leur indice, dans un dictionnaire, les éléments sont identifiés à travers des clés (qui se présente généralement sous forme de chaîne de caractère). Par ailleurs, comme on peut s'attendre, un dictionnaire est une collection d'objets plus générale qui peut inclure à la fois des listes, des tuples ou des array.

Dans cette section, nous allons passer en revue quelques-unes des propriétés des objets dictionnaire.

1.9.1. Définition d'un dictionnaire

Un dictionnaire (ou bibliothèque) est une collection d'objets formée de couples clé-valeur séparés par des virgules, l'ensemble spécifiée à l'intérieur des accolades. Les exemples ci-dessous montrent quelques définitions de dictionnaires

```
x= {'nom':'Jean', 'age':25, 'poids':70, 'taille':1.75}
y= {'Jean':[25,70,1.75],'Paul':[30,65,1.80],'Pierre':[35,75,1.65] }
z= {'Jean':(25,70,1.75),'Paul':(30,65,1.80),'Pierre':(35,75,1.65) }
k={'nom':'Jean','biométrie':[25,70,1.75],'score':(12,17,15),'rang':25}
```

Les quatre variables ci-dessus représentent quatre manières typiques de définition d'un dictionnaire.

La variable x est un dictionnaire dont les clés sont nom, age, poids et taille. Les valeurs correspondantes sont Jean, 25, 70 et 1.75. Remarquons simplement que dans cette définition de x, à chaque clé correspond une valeur unique telle la définition d'une variable simple. Toutefois, il arrive très fréquemment d'associer plusieurs valeurs à une seule clé. C'est le cas de la variable y. En effet, pour cette variable, les clés sont Jean, Paul, et Pierre. Et on associe à chacune de ces clés trois valeurs se présentant sous forme de listes correspondant respectivement à l'âge, le poids et la taille. Quant à la variable z, elle se présente comme la variable y à la seule différence que les valeurs sont spécifiées sous formes de tuples.

Enfin, concernant la variable k, sa structure démontre qu'un dictionnaire peut être constitué de plusieurs types d'objets. En effet, à la clé nom correspond une seule valeur (Jean). Il s'agit d'une variable simple. Pour la clé biométrie, les valeurs correspondantes sont spécifiées sous forme de liste (représentant respectivement l'âge, le poids et la taille), pour la clé score correspond également trois valeurs mais spécifiée sous forme de tuples (représentant les scores obtenus par Jean à trois évaluation successives) et pour la clé rang, on a une valeur unique numérique représentant le rang de Jean à la suite des trois examens qu'il a passés.

Tous ces exemples montrent donc la généralité des objets dictionnaires par rapport aux autres objets tels que les listes, les tuples, les arrays, etc...

1.9.2. Accéder aux éléments d'un dictionnaire (slicing)

Pour accéder aux éléments d'un dictionnaire on se sert des clés. Toutefois la méthode de slicing diffère selon que la clé soit constituée d'une valeur unique ou de plusieurs valeurs comme une liste, tuples, etc...

1.9.2.1. Slicing d'un dictionnaire dans le cas d'une clé à valeur unique

Soit le dictionnaire défini comme suit :

```
x= {'nom':'Jean', 'age':25, 'poids':70, 'taille':1.75}
```

Les clés de x étant définies par des valeurs uniques, on peut faire les slicing suivants

```
x['nom'] # renvoie Jean
x['age'] # renvoie 25
x['taille'] # renvoie 1.75
```

Pour faire du slicing dans le cas d'un dictionnaire dont les clés sont à valeur unique, il suffit d'indiquer la clé dans l'opérateur de slicing [ ] à l'image des listes ou tuples pour lesquelles il faut indiquer les indices.

1.9.2.2. Slicing d'un dictionnaire dans le cas d'une clé à plusieurs valeurs

Dans le cas d'un dictionnaire dont les clés sont à valeur multiples, on distingue deux situations : les slicing de premier niveau et les slicing de second niveau. Dans le premier cas, on accède à l'ensemble des éléments qui correspondent à la clé et dans le second cas, on accède à un élément particulier de l'ensemble qui forme la clé. Les exemples ci-dessous sont des illustrations.

Soit le dictionnaire x défini comme suit :

```
x= {'Jean':[25,70,1.75],'Paul':[30,65,1.80],'Pierre':[35,75,1.65] }
```

On peut faire des slicing suivants :

```
x['Jean'] #renvoie [25, 70, 1.75]
x['Jean'][0] #renvoie 25
x['Jean'][0 :2] #renvoie [25, 70]
```

Ces exemples montrent que pour accéder aux sous-éléments il faut ajouter à la clé un opérateur de slicing supplémentaire dont les arguments sont, cette fois, les indices des éléments dans la séquence de valeurs correspondant à la clé.

1.9.3. Modification de dictionnaire : ajout et suppression de clés ou de valeurs

On peut modifier un dictionnaire soit en modifiant les clés et les valeurs déjà existantes ou en ajouter des nouvelles. On peut également supprimer des valeurs mais aussi des clés. Les exemples ci-dessous illustrent ces cas.

1.9.3.1. Ajout ou modification de clés ou de valeurs

Soit le dictionnaire vide x défini comme suit :

```
x= {} #  Crée un dictionnaire vide. On pouvait aussi utiliser x=dict()
```

On va ajouter des clés et des valeurs à x et essayer d'apporter des modifcations au dictionnaire

```
x['nom'] = 'Jean' # ajoute la clé-valeur nom et Jean au dictionnaire x initial
x['biométrie']=  [25,70,1.75]  # ajoute la clé-valeur biométrie et [25,70,1.75] au dictionnaire x
x['biométrie']= [30,70,1.80] # modifie les valeurs de la clé biométrie en effectuant une nouvelle définition
x['biométrie'][0]= 2 # modifie l'élément d'indice 0 de la liste de valeur qui correspond à la clé biométrie (préalabelemnt définie)
```

1.9.3.2. Suppression de clés ou de valeurs dans un dictionnaire

Il est possible de supprimer des clés ou des valeurs dans un dictionnaire. Les exemples ci-dessous sont des illustrations.

Soit le dictionnaire x défini comme suit :

```
x= {'Jean':[25,70,1.75],'Paul':[30,65,1.80],'Pierre':[35,75,1.65] }
```

On peut faire des opérations de suppression suivantes :

```
del  x['Jean']  # Supprime la clé Jean toutes les valeurs correspondantes
del x['Jean'][0] # Supprime l'élément d'indice 0 dans la séquence de valeur correspondant à la clé Jean. Pour une clé à une seule valeur, on utilise del x['nomCle'] où nomCle est le nom de la clé qui a la valeur unique.
```

1.9.4. Renommer une clé dans un dictionnaire

Pour renommer une clé dans un dictionnaire, on utilise la fonction pop() telle que définie dans l'exemple suivant :

Soit le dictionnaire x défini comme suit :

```
x= {'Jean':[25,70,1.75],'Paul':[30,65,1.80],'Pierre':[35,75,1.65] }
```

On souhaite renommer la clé Jean comme John. Alors on utilise la méthode de renommage comme suit :

```
x['John']=x.pop('Jean')
```

1.9.5. Tester l'existence d'une clé dans un dictionnaire : la fonction in

Pour vérifier si une clé existe dans un dictionnaire, on utilise la fonction in qui renvoie une valeur booléenne (True ou False) selon que la clé existe ou pas dans la liste. Exemple : soit le dictionnaire x définir par :

```
x= {'Jean':[25,70,1.75],'Paul':[30,65,1.80],'Pierre':[35,75,1.65] }
```

On peut faire des tests d'existence suivants :

```
'Paul' in x # renvoie True (car il y a bien Paul parmi les clés de x)
'Sebastien' in x # renvoie False
'Sebastien' not in x # renvoie True
'Jean' not in x # renvoie False
```

Nb : Encore une fois, la fonction « in » peut s'avérer utile dans de nombreuses situations. Elle peut par exemple servir dans la définition des instructions conditionnelles sur les clés en l'associant avec la clause « if ». Exemple :

```
if 'Paul' in x :
     print("Oui, Paul est bien une clé dans le dictionnaire x")
else :
     print("Non,  Paul n'est pas une clé dans le dictionnaire x")
```

1.9.6. Récupérer les clés ou les valeurs d'un dictionnaire (les fonctions keys() et values())

Dans de nombreuses situations, il arrive qu'on ait besoin de récupérer les clés ou les valeurs d'un dictionnaire pour d'autres utilisations dans le programme (ex : constituer des listes, ou des tuples, etc...). Dans ces cas, on doit d'abord récupérer les clés et les valeurs. Pour cela, on utilise les fonction keys() et values().

La fonction keys() renvoie les clés d'un dictionnaire alors que la fonction values renvoie les valeurs. Les exemples ci-dessous montrent comment utiliser ces deux fonctions.

Soit le dictionnaire x défini comme suit :

```
x= {'Jean':[25,70,1.75],'Paul':[30,65,1.80],'Pierre':[35,75,1.65] }
```

On peut récupérer les clés et les valeurs comme suit :

```
myKeys=x.keys() # renvoie dict_keys(['Pierre', 'Paul', 'Jean'])
myValues=x.values() # renvoie dict_values([[35, 75, 1.65], [30, 65, 1.8], [25, 70, 1.75]])
```

Signalons alors simplement qu'il est possible de transformer ces objets en objets liste en utilisant la fonction list() comme suit :

```
myKeysList=list(myKeys) # renvoie ['Pierre', 'Paul', 'Jean']
myValuesList=list(myValues) # renvoie [[35, 75, 1.65], [30, 65, 1.8], [25, 70, 1.75]]
```

1.10. Etude des fonctions

Simplement définie, une fonction est un bout de programme, un ensemble d'instructions agencées dans le but de réaliser une ou plusieurs tâches bien définies. On distingue deux catégories de fonctions sous python, les fonctions prédéfinies et les fonctions écrites par les utilisateurs. Les fonctions prédéfinies sont des fonctions directement intégrées dans la bibliothèque standard du système python alors que les fonctions-utilisateurs sont écrites soit par l'utilisateur actuel ou par d'autres utilisateurs. Dans cette section, nous allons étudier comment élaborer et appliquer une fonction sous python. Mais avant cela, nous donnerons un aperçu général sur quelques fonctions de base sous python.

1.10.1. Aperçu général sur quelques fonctions prédéfinies sous python : la fonction print() et la fonction input()

La plupart des tâches de programmations réalisées sous python par les utilisateurs se basent d'abord en priorité sur une multitude de fonctions préinstallées sous python et qui font la richesse de ce langage. L'objectif de cette section n'est pas de passer en revue toutes les fonctions disponibles sous python (nous n'en n'aurons pas les moyens !). Il s'agit simplement de présenter brièvement deux fonctions de base qui sont incontournables dans la programmation python. Il s'agit de la fonction print() et de la fonction input()

1.10.1.1. La fonction print()

Comme on le sait déjà, la fonction print() a pour rôle d'afficher à l'écran les valeurs des objets spécifiés comme arguments. Exemple :

```
print("Bonjour", "à", "tous")
x=12
print(x)
y=[1, "lundi","12", 5, 3, "valeur test"]
print(y)
```

On peut remplacer le séparateur par défaut (l'espace) par un autre caractère quelconque (ou même par aucun caractère), grâce à l'argument sep.

```
print("Bonjour", "à", "tous", sep= "****")
print("Bonjour", "à", "tous", sep= "")
```

Nb : N'oublier pas aussi qu'au lieu d'utiliser la fonction print() avec plusieurs argument, on peut utiliser l'opérateur de formatage "%" ou la fonction format() afin d'afficher le message en un seul bloc (se référer à l'utilisation de ces opérateur dans les sections plus haut). Exemple :

```
x=42
print("La distance à parcourir dans un marathon est de %i km" %x) # renvoie La distance à parcourir dans un marathon est de 42 km
```

1.10.1.2. La fonction input()

La fonction input permet de donner la main à l'utilisateur afin qu'il saisisse la valeur d'un argument donné

Exemple 1 :

```
prenom = input("Entrez votre prénom : ")
print("Bonjour,", prenom)
```

Exemple 2

```
print("Veuillez entrer un nombre positif quelconque : ")
ch = input()
num = int(ch) # conversion de la chaîne en un nombre entier
print("Le carré de", num, "vaut", num**2)
```

Nb : Soulignons que la fonction input() renvoie toujours une chaîne de caractères. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée (qui sera donc de toute façon de type string) en une valeur numérique du type qui vous convient, par l'intermédiaire des fonctions intégrées int() (si vous attendez un entier) ou float() (si vous attendez un réel).

1.10.2. Les fonctions définies par l'utilisateur

Les fonctions-utilisateurs sont des bouts de programme écrits par l'utilisateur afin de réaliser des tâches bien définies. Rappelons simplement que tout programme vise à résoudre un problème donné. Il est donc essentiel que la problématique à laquelle doit répondre le programme soit clairement définie et que les instructions qui concourent à la résolution soient aussi clairement spécifiées afin de pouvoir les traduire correctement en langage informatique. Le but de cette section est d'étudier comment élaborer une fonction.

1.10.2.1. Définir une fonction

Pour définir une fonction sous Python, on utilise le mot clé « def » pour déclarer le nom de la fonction. La syntaxe générale de définition d'une fonction est la suivante : :

```
def nomDeLaFonction(arg1, arg2, …, argN):
    bloc d'instructions
```

def est le mot clé qui déclare le nom de la fonction. Ce mot est un mot réservé de python qui figure parmi les 33 mots réservés du langage python. Ce qui signifie qu'aucun nom d'objet ne doit prendre cette valeur. Il est utilisé uniquement pour définir une fonction.

A la suite du mot clé def, on indique le nom que doit prendre la fonction. Comme dans le cas des noms de variables, il est conseillé d'utiliser surtout des lettres minuscules, notamment au

début du nom. Et pour des noms composés, on peut utiliser des majuscules pour faire la transition entre les éléments qui composent le nom (ex : myFunction, fonctionTest, etc...). Notons simplement que les noms d'objets commençant par une majuscule seront réservés au cas des classes.

Après avoir déclaré le nom de la fonction, on indique entre parenthèses les noms des arguments (paramètres) à partir desquels seront définies les instructions dans le corps du programme. Notons toutefois que la déclaration des noms des paramètres n'est pas obligatoire car il peut exister des programmes sans arguments ou du moins ceux-ci seront déclarés directement dans le corps du programme. Quoiqu'il en soit, la spécification des parenthèses reste obligatoire même si aucun paramètre n'est déclaré à l'intérieur. Attention également au symbole « : » qui marque la fin de la déclaration de la fonction et le début de la définition des instructions tout comme dans le cas de l'utilisation de la fonction « if » ou des boucles « for » ou « while ».

Après avoir déclaré la fonction à travers son nom et ses arguments, la seconde étape est la spécification des blocs d'instructions. Ceux-ci doivent être définis avec un certain décalage par rapport au mot clé def (indentation) tout comme dans le cas de la définition de la fonction « if » ou des boucles « for » ou « while ». Cette indentation des blocs d'instructions est obligatoire.

Dans cette section, nous allons étudier plusieurs variantes de la définition de fonctions notamment les fonctions définies sans arguments, les fonctions avec arguments se présentant sous forme de paramètres (constants), les fonctions avec arguments se présentant sous forme de variables, etc...

1.10.2.1.1. Définition d'une fonction simple sans argument

L'exemple ci-dessous illustre la définition d'une fonction simple sans argument. Le but de la fonction est d'afficher les 20 premières valeurs de la table de multiplication par 8.

Ainsi, on a :

```
def tableMultiplication8():
      n = 1
      while n <=20 :
            v=n*8
            print(n, 'x', 8, '=', v, sep =' ')
            n = n +1
```

Quelques petites remarques peuvent être faites sur cette fonction. D'abord, la fonction est construite sur une boucle « while » définie à partir d'une variable n dont la valeur initiale est 1 et qui sera incrémentée jusqu'à ce que sa valeur atteigne 20. Ensuite, pour chaque valeur de n, on définit une variable v dont la valeur est le produit de 8 et n. Enfin, on affiche un ensemble de chaîne de caractères défini à l'intérieur de la fonction print().

Remarquons simplement que la variable n a été initialisée à l'extérieur de la boucle mais son incrémentation s'effectue de la boucle à la suite des blocs d'instructions spécifiés à l'intérieur de la boucle. On peut aussi remarquer deux niveaux d'indentation. Un premier niveau d'indentation est définie à partir de la déclaration def et un second niveau d'indentation est défini à la suite de la déclaration « while ». Cette règle reste valable dans tous les cas. Tout bloc d'instruction défini à la suite d'une déclaration def sera indenté. Et lorsque ce bloc d'instruction est formé d'une clause « if » ou d'une boucle « while » ou « for », les instructions appartenant à ces blocs serons indentées à leur tour et cela de façon séquentielle.

Pour exécuter la fonction tableMultiplication8() que nous venons de définir, il suffit de le référencer en indiquant son nom comme suit (n'importe où dans le programme principal).

```
tableMultiplication8() # appelle la fonction tableMultiplication8()
```

Signalons aussi qu'on pouvait aussi définir la fonction tableMultiplication8() à partir d'une boucle « for ». Dans ce cas, la fonction se présenterait comme suit :

```
def tableMultiplication8():
      for n in range(1 ,21) :
            v=n*8
            print(n, 'x', 8, '=', v, sep =' ')

tableMultiplication8() # appelle la fonction tableMultiplication8()
```

Ici, on utilise la fonction range(1, 21) qui renvoie une séquence de valeur allant de 1 à 20. Chaque élément de cette séquence est récupérée et mutilplié par 8 avec la boucle « for ». Cet exemple montre donc que la boucle « while » et « for » sont alternatives dans de nombreuses situations.

1.10.2.1.2. Définition d'une fonction dont les arguments sont des paramètres

Un paramètre est une variable qui prend une valeur constante. Dans l'exemple précédent, nous avons élaboré une table de multiplication par 8. Nous pouvons généraliser cette fonction de sorte que qu'elle puisse renvoyée la table de multiplication de n'importe quel nombre spécifié comme argument. Ces nombres étant des paramètres, il s'agit alors de définir une fonction dont les arguments sont les paramètres. Voir l'exemple ci-dessous

```
def tableMultiplication(base):
      n = 1
      while n <=20 :
            v=n*base
            print(n, 'x', base, '=', v, sep =' ')
            n = n +1
```

```
tableMultiplication(2) # renvoie la table de multiplication par 2
tableMultiplication(8) # renvoie la table de multiplication par 8
tableMultiplication(11) # renvoie la table de multiplication par 11
```

1.10.2.1.3. Définition d'une fonction dont les arguments sont des variables

Une fonction dont les arguments sont des variables est une fonction dont les arguments sont des paramètres à la seule différence que dans le premier cas l'argument plusieurs valeurs successivement lors de l'appel du programme. Exemple : Considérons la fonction tableMutplication() qui a été définie précédemment comme suit :

```
def tableMultiplication(base):
      n = 1
      while n <=20 :
            v=n*base
            print(n, 'x', base, '=', v, sep =' ')
            n = n +1
```

Pour que l'argument base se comporte comme une variable, on va l'attribuer plusieurs valeurs à l'intérieur d'une nouvelle boucle « while » ou « for ».

Exemple : réalisons la table de multiplication des 20 premières valeurs de tous les nombres allant de 2 à 10.

On a :

```
x = 2
while x <=10:
    tableMultiplication(x)
    print(" ")  # insérer un espace entre deux tables
    x = x +1
```

Dans cet exemple, l'argument base devient simplement une variable qui prend les valeurs de (définie de 2 à 10).

1.10.2.1.4. Définition d'une fonction à plusieurs arguments

Dans les exemples précédents, les fonctions ont été définies à partir d'un argument unique (ou sans argument). Une fonction est généralement définie à partir de plusieurs arguments comme l'a montré que la syntaxe générale en début de section. Ces arguments peuvent être des paramètres ou des variables. Certaines peuvent être des arguments obligatoires et d'autres optionnels. Par ailleurs, lorsque la fonction est définie à partir de plusieurs arguments, ceux-ci doivent être séparés par des virgules.

Exemple : la fonction tableMultiplication () définie ci-dessus effectue la multiplication en considérant les 20 premiers éléments de la base (1 à 20). On peut généraliser cet intervalle en indiquant les valeurs de début et de fin en ajoutant deux argument supplémentaires comme suit :

```
def tableMultiplication(base, debut, fin):
      n = debut
      while n <=fin :
            v=n*base
            print(n, 'x', base, '=', v, sep =' ')
            n = n +1
```

Pour exécuter la fonction, on a :

```
tableMultiplication(10, 5, 20) # effectue la table de mutplication de 10 allant de 5 à 20
tableMultiplication(8, 6, 12) # effectue la table de mutplication de 8 allant de 6 à 12
```

NB : Dans cet exemple, base, debut et fin sont spécifiés comme paramètres. On peut aussi les traiter comme des variables en élaborant par exemple une boucle lors de l'exécution de la fonction. Exemple :

```
x = 2
y=5
z=20
while x <=10:
    tableMultiplication(x ,y, z)
    print(" ")  # insérer un espace entre deux tables
    x = x +1
```

Cette boucle renvoie la table de multiplication de 2 à 10 (pour les éléments allant de 5 à 20).

1.10.2.1.5. Définition des valeurs par défaut pour les arguments d'une fonction

Lors de la définition d'une fonction, il est souvent conseillé de définir des valeurs par défaut pour certain arguments (notamment les arguments optionnels).

En effet, il est conseillé de rendre obligatoire certains arguments du programme et de renvoyer des messages d'erreur lorsque les valeurs ne remplissent pas un certain nombre de conditions bien définies (par exemple, l'argument ne doit pas être NULL, on ne doit pas utiliser une variable en caractère là une variable numérique est prévue, etc…). En revanche, on peut définir des valeurs par défaut pour des des arguments optionnels à condition que ces valeurs soient universelles c'est-à-dire valable quel que soit le contexte dans lequel le programme est exécuté..

En définissant les valeurs par défaut pour les arguments d'une fonction, il est possible d'appeler le programme avec une partie seulement des arguments attendus. Exemples :

```
def salutation(nom, titre ='Monsieur'):
```

```
    print("Bonjour", titre, nom)
```

La fonction salutation ainsi définie a deux arguments : nom et titre. Une valeur par défaut a été définie pour l'argument. Ainsi lorsque la fonction salutation est appelée avec seulement l'argument nom (omettant l'argument), la fonction renvoie la valeur par défaut Monsieur. Exemple :

```
salutation('Dupont') # renvoie Bonjour Monsieur Dupont
```

Mais lorsque la fonction est appelée avec les deux arguments, la valeur par défaut est ignorée. Exemple :

En définissant les valeurs par défaut pour les arguments d'une fonction, il est possible d'appeler le programme avec une partie seulement des arguments attendus. Exemples :

```
salutation('Dupont', 'Mademoiselle')
```

NB : Les arguments sans valeur par défaut doivent être spécifiés avant les arguments avec les valeurs par défaut. Par exemple, la définition ci-dessous est incorrecte et renvoie une erreur lors de l'exécution.

```
def salutation(titre='Monsieur', nom):
```

1.10.2.1.6. Ordre des arguments lors de l'appel d'une fonction

Dans la plupart des langages de programmation, lors de l'appel d'une fonction les arguments doivent être spécifiés exactement dans le même ordre que celui dans lequel ils ont été indiqués lors de la définition de la fonction. Python fait cependant exception à cette règle en autorisant une souplesse beaucoup plus grande. En effet, on peut faire appel à la fonction en fournissant les arguments correspondants dans n'importe quel ordre, à condition de spécifier nommément les paramètres correspondants tout en assignant les valeurs. Exemple :

```
def salutation(nom, titre):
    print("Bonjour", titre, nom)
```

Pour appeler cette fonction dans un ordre quelconque on fait :

```
salutation(titre='Mademoiselle', nom='Dupont') # renvoie Bonjour Mademoiselle Dupont
salutation(nom='Dupont', titre='Mademoiselle') # renvoie Bonjour Mademoiselle Dupont
salutation( nom='Dupont', titre='Monsieur') # renvoie Bonjour Monsieur Dupont
```

Ces exemples montrent que l'ordre de spécification des arguments lors de l'appel de la fonction n'est pas importante tant que les noms et les valeurs des arguments sont explicitement indiquées. Cette règle reste également valable lorsque les paramètres ont reçu tous une valeur par défaut, sous la forme déjà décrite ci-dessus. Cependant, l'ordre de définition doit être respecté dans le cas où certains arguments ont des valeurs par défaut et d'autres non. Il faut juste rappeler que les arguments sans valeurs par défaut doivent être déclarés avant les arguments avec valeurs par défaut

1.10.2.2. Les fonctions lambda

Une fonction lambda est une fonction anonyme c'est-à-dire une fonction constituée d'un bloc d'instructions appelable et réutilisable comme une fonction, mais sans nom. Une fonction lambda est généralement utilisée pour des fonctions très courtes avec peu d'instructions (nécessitant pas alors d'écrire une fonction classique avec le mot clé def et un appel)

La syntaxe générale de définition d'une fonction lamda est la suivante:

```
 lambda arg1, arg2,…, argN : bloc instructions(ou formules)
```

Une fonction lambda est déclarée avec le mot clé lambda qui figure par les 33 mots réservés de python. Ce mot clé est accompagné par le (ou les arguments) qui entrent dans la définition de la fonction. La fin de la déclaration est marquée par le symbole « : » à la suite duquel on spécifie les instructions. L'exemple ci-dessus illustre la défintion d'une fonction lambda.

```
lambda x, y : x * y
```

On constate que contrairement à une fonction classique, la fonction lambda n'est définie avec aucun nom. Les arguments ne sont pas obligatoirement spécifiés à l'intérieur de parenthèses.

La fonction ci-dessus a été définie avec deux arguments x et y et l'instruction qui constitue la fonction est définie comme le produit entre x et y.

Notons toutefois que même si la fonction lambda n'est pas définie avec un nom, pour récupérer la valeur renvoyée, lors de l'appel de la fonction, il faut l'assigner à une nouvelle variable. L'exemple ci-dessous illustre l'appel de la fonction lambda précédente en prenant x=2 et y=3.

```
x=lambda x, y : x * y
x(2,3) # appelle la fonction avec x=2 et y =3; renvoie 2*3=6
```

1.10.3. Les variables locales et les variables globales

Lorsque nous définissons des variables à l'intérieur d'une fonction, ces variables ne sont accessibles que pour cette fonction elle-même. On dit que ces variables sont des variables « locales » à la fonction. Mais lorsque les variables sont définies à l'extérieur de la fonction dans le corps du programme principal. Ces variables sont appelées variables « globales ». Le

contenu d'une variable globale est visible et accessible à partir de l'intérieur d'une fonction, mais cette fonction ne peut pas modifier la valeur de la variable. Les exemples suivants permettent d'illustrer ce qui distingue une variable locale d'une variable globale.

Soit la fonction myFunction définie comme suit :

```
def myFunction():
    p = 20
    print(p, q)
p=15
q=38
print(p, q) # renvoie  15 38
myFunction() # appelle la fonction, renvoie 20 38
print(p, q) # renvoie  15 38
```

On constate en effet que lorsque la fonction myFunction() est appelée, la variable globale q y est accessible et sa valeur est correctement renvoyée. En revanche, pour la variable p, nous avons deux définitions. Une première définition effectuée à l'intérieur de la fonctioon myFunction() et une seconde définition effectuée dans le programme principal à l'extérieur de la fonction myFunction(). Dans la première définition, p se comporte comme une variable locale et dans la deuxième, p est une variable globale. On constate que c'est la valeur de la variable locale qui est renvoyée et la définition de la variable locale n'a aucune incidence sur la variable globale. Au final, on retient donc qu'une fonction renverra toujours la valeur d'une variable en priorité par rapport à une variable globale mais la valeur de cette dernière ne sera pas modifiée.

Cependant, on peut modifier ce comportement par défaut en permettant à la fonction de modifier la valeur de la variable globale. Dans ce cas, il faut explicitement définir la variable comme global à l'intérieur de la fonction. Exemple:

```
def myFunction():
    global p
    p = 20
    print(p, q)
p=15
q=38
print(p, q) # renvoie  15 38
myFunction() # appelle la fonction, renvoie 20 38
print(p, q) # renvoie  20 38
```

Dans cet exemple, la variable p a été déclarée comme global à l'intérieur de la fonction. Par conséquent, le programme peut modifier cette variable à tout moment lors de l'appel. La preuve est que dans la fonction print() utilisée avant l'appel de la fonction, les valeurs de p et de q définies dans le programme sont affichées. En appelant la fonction, celle-ci affiche la valeur de p définie à l'intérieur de la fonction. Cette valeur remplace d'ailleurs systématiquement la

précédente valeur définie à l'extérieur de la fonction. Cela se voit en utilisant la fonction print() une seconde fois à après l'appel de la fonction. Les valeurs de p devient maintenant 20. Au final, on retient qu'une fonction Python ne permet pas la modification d'une variable globale que lorsque cette variable est déclarée comme globale à l'intérieur de la fonction.

1.10.4. Récupérer la valeur renvoyée par une fonction : l'instruction return

Très fréquemment, il arrive qu'on ait besoin de récupérer pour d'autres utilisations le résultat final renvoyé par l'appel d'une fonction. Le résultat final renvoyé par une fonction est un objet qui peut être de différents types (variable simple, liste, tuples, etc...). Pour récupérer ces valeurs, on utilise l'instruction « return » qui renvoie la valeur de l'instruction à laquelle elle est rattachée. D'une manière générale, l'instruction return est spécifiée à l'intérieur d'une fonction qu'il s'agisse d'une fonction classique ou d'une fonction lambda.

Les exemples ci-dessous illustrent l'utilisation de l'instruction return dans différentes situations.

Exemple 1 : Cas où l'on renvoie une valeur unique

Soit la fonction ci-dessous qui calcule le cube d'un nombre telle que :

```
def calculCube(x):
      cube=x**3
      print(cube)
```

On souhaite récupérer la valeur de cube et la stocker dans une nouvelle variable nommée x. Alors, on peut ajouter l'instruction return à la fonction comme suit :

```
def calculCube(x):
      cube=x**3
      print(cube)
      return cube # renvoie la valeur de cube
```

Pour récupérer la valeur de cube dans la variable x on procède par assignation comme suit :

```
x=calculCube(5) # Calcul le cube de 5 et renvoie sa valeur (125) dans x.
print(x) # renvoie 125
```

L'instruction return est nécessaire pour définir la variable x sinon celle-ci contiendra la valeur None.

Exemple 2 : Cas où l'on renvoie un ensemble de valeurs (ex : une liste)

On souhaite récupérer les 10 premières valeurs de la table de multiplication d'un nombre quelconque en stockant ces résultats dans une liste. Alors, on suit la méthode suivante :

```python
def tableMultiplication(base):
    listeValeurs = [] # Définir une liste vide qui doit accueillir les
résultats
    n = 1 # initialisation de n à 1
    while n <=10:
        v = n * base
        listeValeurs.append(v) # ajout du terme à la liste vide
        n = n +1 # incrémentation de n
    return listeValeurs
```

Exécutons cette fonction avec la table de multiplication par 9

```python
x = tableMultiplication(9) # Appelle la fonction avec l'argument 9
print(x) # renvoie [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

1.10.5. Utilisation des générateurs de fonction : l'instruction yield

« yield » est un mot clé qui représente une instruction de même type que le mot clé « return », à la seule différence que return renvoie une valeur ou un ensemble de valeurs alors que l'instruction yield renvoie un générateur.

Pour mieux comprendre la logique de l'instruction yield, prenons un exemple concret comme celui-ci.

```python
def myFunction1() :
    listeValeur=[]
    for i in range(2, 11):
        listeValeur.append(i**2)
    return listeValeur
x=myFunction1()
print(x) #renvoie [4, 9, 16, 25, 36, 49, 64, 81, 100]
```

myFunction1 est une fonction classique qui permet de renvoyer (avec l'instruction return) la lsite des carrés de tous les éléments appartenant à la séquence de valeurs comprise entre 2 et 10

Proposons une nouvelle définition de la fonction myFunction en utilisant l'instruction yield. On a :

```python
def myFunction2() :
    for i in range(2, 11):
        yield i**2
myGenerator=myFunction2()
print(myGenerator)   #renvoie   <generator   object   myFunction2   at
0x0B3D9D78>
```

D'abord, la première différence qui transparait avec l'utilisation de l'instruction return c'est que l'instruction yield renvoie un objet générateur. Par conséquent un simple appel de la fonction myFunction2 ne permet d'afficher les valeurs renvoyées par l'instruction yield.

Dans l'exemple ci-dessus puisque l'instruction yield est définie à l'intérieur d'une boucle, on doit alors récupérer les valeurs dans le générateur (nommé ici myGenerator) une à une en élaborant une nouvelle boucle comme suit :

```
x=[] # Définir une liste vide qui doit accueillir les résultats
for i in myGenerator :
        x.append(i)
print(x) #renvoie [4, 9, 16, 25, 36, 49, 64, 81, 100]
```

On remarque alors que l'instruction return renvoie la liste finale alors que l'instruction yield définit d'abord les valeurs et les stocke dans un objet de type générateur. On récupère alors les éléments en utilisant une boucle pour former la liste finale des valeurs qui constitue les résultats. Cet aspect représente la principale particularité de l'instruction yield par rapport à l'instruction return.

Cependant, la présentation qui vient d'être faite de l'instruction yield est une présentation un peu sommaire et simpliste. L'instruction yield est un peu plus subtile en réalité. En effet, en toute rigueur conceptuelle, lorsqu'une fonction est appelée, l'instruction yield sert à suspendre l'exécution de la fonction chaque fois qu'une instruction yield est rencontrée. L'exécution de la fonction reprend alors là où elle s'était arrêtée lorsque la fonction est appelée une nouvelle fois et sera encore suspendue lorsqu'une nouvelle instruction yield est rencontrée. Ainsi de suite. Considérons par exemple la fonction suivante.

```
def myFunction():
    yield "Ceci est une chaîne de caractères"
    yield "Oui bien sûr"
    yield 1200

myGenerator=myFunction() #appel de la fonction et son assignation à une variable nommée myGenerator
```

Cette fonction contient trois spécifications de l'instruction yield. Cela signifie que pour un premier appel de la fonction, celle-ci ne va renvoyer que la valeur "Ceci est une chaîne de caractères" car l'exécution de la fonction est suspendue à chaque fois que l'instruction yield est rencontrée. Et pour un second appel, l'exécution reprend là où elle s'était arrêtée. En effet, Python sauvegarde l'état du code du générateur entre chaque appel. Au final, pour pouvoir renvoyer tous les valeurs dans une fonction il faut spécifier une boucle dont le nombre d'itérateurs est égal au nombre de fois où la fonction yield est répétée dans la fonction. D'où la nécessité d'utiliser une boucle « for.. in » afin que le code puisse être exécuté autant de fois que de yield. Par exemple, pour afficher les trois valeurs de la fonction myFunction(), on procèdera comme suit :

```
for i in myGenerator :
        print(i)
```

1.10.6. Gestion des erreurs et des exceptions lors de la définition d'une fonction

1.10.6.1. Utilisation des blocs ty-except

Lors de l'appel d'une fonction, l'interpréteur Python peut rencontrer des instructions qu'il n'est pas en mesure d'exécuter (ex : erreur de syntaxe, division par zéro, accès à un fichier qui n'existe pas, etc). Dans ces cas, il jette une exception et renvoie un message d'erreur. Par exemple, soit l'instruction définie comme suit :

```
val =float(input("Veuillez saisir un nombre :"))
```

Dans cette instruction, on passe la main à l'utilisateur en lui demandant de saisir un nombre. Et dans le cas où l'utilisateur saisit des chaînes de caractères contenant des valeurs alphabétiques ou des caractères spéciaux, python enverra un message d'erreur de type « *ValueError: could not convert string to float...* » car la fonction float() ne permet de convertir que des chaînes constituer des valeurs en chiffres. Dans cette situation, on est face à une exception. Si cette exception n'est pas gérée par le programme dont l'instruction fait partie, le résultat est l'arrêt du programme avec affichage d'un message d'erreur.

Mais, le langage python offre les possibilités au programmeur pour contrôler les instructions pouvant générer des exceptions et de définir ses propres traitements d'erreur. Ces traitements sont basés sur l'utilisation des mots clés « try » et « except » qui figurent également parmi les 33 mots réservés de python. Les instructions susceptibles de générer des exceptions sont placées à l'intérieur d'un bloc « try » et les instructions à exécuter en cas d'exceptions sont définies à l'intérieur d'un bloc « except ».

Un bloc try doit être accompagné d'au moins un bloc except. Plusieurs blocs except peuvent être associés à un même bloc try. Ainsi, un ensemble try-except s'écrira de la forme :

```
try:
      instructions
except ErrorType1:
      instructions si ErrorType1
except ErrorType2:
      instructions si ErrorType2
except (ErrorType3, ErrorTpe4):
      instructions si ErrorType3 ou ErrorType4
except:
      instructions si autre erreur
```

Il existe trois manières d'écrire la ligne except :

1- en indiquant le nom de l'erreur concernée : ex :

```
except Error:
```

2- en indiquant un tuple contenant plusieurs erreurs : ex :

```
except (Error1, ..., Errorn):
```

3- en n'indiquant rien :

```
except:
```

Les exemples présentés ci-après illustrent l'utilisation de des blocs try et except.

Exemple1: Demandons à l'utilisateur de saisir des valeurs et lui redemander une nouvelle fois la saisie afin qu'il entre une valeur convenable

```
try :
    val =float(input("Veuillez saisir un nombre :"))
except ValueError:
    print ("La valeur entrée n'est pas valide")
    val =float(input("Veuillez recommencer s'il vous plaît:"))
```

Notons ici l'erreur renvoyée en cas d'échec de la saisie est de type ValueError. D'où la présence de ce mot clé devant l'instruction. Bien sûr, il existe plusieurs autres types d'erreurs identifiables par des codes d'erreur qui permettent de les distinguer et d'effectuer les traitements adéquats.

Dans cet exemple, l'instruction except renvoie d'abord un message indiquant à l'utilisateur que la valeur saisie n'est pas valide. Ensuite, il lui passe la main une seconde fois pour qu'il saisisse une nouvelle fois.

Noter toutefois que cette structure de code ne donne la main à l'utilisateur qu'un fois à l'issue duquel si une valeur valide n'est pas saisie, le programme s'arrêtera. Mais on peut améliorer le code de sorte que la fonction donne la main à l'utilisateur autant de fois que nécessaire jusqu'à ce qu'il saisisse une valeur valide. L'exemple ci-dessous est une illustration.

```
test = "Pas Ok"
while (test !="Ok") :
     try :
         val =float(input("Veuillez saisir un nombre :"))
         test="Ok"
     except ValueError:
```

```
    print  ("La  valeur  entrée  n’est  pas  valide,  Veuillez
recommenccer s’il vous plaît")
```

Dans cette formulation, on définit d’abord une variable nommée *test* dont la valeur est "Pas Ok". Ensuite, on construit une boucle conditionnelle qui évalue chaque fois la valeur de la variable *test*. Tant que la valeur de la variable *test* est différente de «Ok », les instructions de la boucle sont exécutées. La boucle a été construite autour de deux blocs d’instructions. Le bloc try dans lequel on demande la saisie de la valeur de la variable *val*. Et lorsque cette saisie se passe correctement, on attribue la valeur « Ok » à test. Mais lorsque la saisie par l’utilisateur ne se passe pas correctement, les instructions du bloc except sont exécutées. Dans le bloc except, il y a l’instruction print() qui donne des indications à l’utilisateur. Une fois que ce message est affiché, la boucle reprend puisque la valeur de test n’a pas été modifiée.

Exemple 2 : Dans le deuxième exemple, on va essayer d’ouvrir un fichier en utilisant la fonction open(). Si le fichier n'est pas trouvé au chemin indiqué la fonction renvoie une exception de type IOError. La gestion de cette exception permet l'affichage d'un message adéquat :

```
try :
      nomFichier = input("Entrez un nom de fichier : ")
      myFichier = open(nomFichier)
except IOError:
      print("Erreur de lecture du fichier")
      sys.exit()
```

Remarque : Il est également possible pour le programmeur de déclencher explicitement une exception grâce à l'instruction *raise <Nom d'une Exception>*, et aussi il est possible de définir de nouvelles exceptions pour des besoins particuliers

Exemple 3 : Utilisation de l’instruction excpt avec l’instruction pass

Soit la liste x définie comme suit:

```
x= ['North', '6.47', '4.03', 'Yorkshire', '6.13', '3.76', 'Northeast',
'6.19', '3.77', 'East', 'Midlands', '4.89', '3.34']
```

Les éléments de cette liste sont tous en caractères. On souhaite alors convertir les élements en chiffres en format numérique en utilisant la fonction float(). Nous choisissons alors d’élaborer un boucle qui scanne tous les éléments un à un tente des convertir. Dans cette situation, il n’est nécessaire de lever une exception qui indique de passer à l’élément suivant lorsque l’élément actuel n’est pas convertible. Pour cela, on élabore la boucle suivante :

```
for i in range(len(x) ):
    try:
        x[i] = float(x[i] )
    except:
        pass
```

L'instruction pass permet de continuer la boucle lorsqu'une chaine de caractère non convertible est rencontrée.

L'instruction « pass » est très proche des instructions « break » et « continue » qui ont été présentées dans les sections précédentes. En effet, nous avons montré que l'instruction « break » arrête la boucle lorsqu'une condition secondaire spécifiée à l'intérieur de la boucle est vérifiée. Quant à l'instruction « continue », il suspend l'exécution des instructions de la boucle lorsqu'une condition secondaire définie à l'intérieur de la boucle est vérifiée. L'exécution de la boucle continue lorsque cette condition n'est plus vérifiée. De ce point de vue, l'instruction « continue » et « pass » sont très proches.

1.10.6.2. Utilisation des clause « else » et « finally » dans une exception

Au même titre que pour les clauses « if », il est possible d'adjoindre un bloc « else » aux blocs de gestion d'exceptions : ce dernier ne sera alors exécuté que si aucune exception n'a été levée. Par ailleurs, il est aussi possible d'ajouter un bloc « finally » à la définition d'une exception. Les instructions de ce dernier bloc seront exécutées qu'il y ait des exceptions ou pas (par exemple instruction à exécuter pour fermer un fichier ouvert par les instructions try, etc...). L'exemple ci-dessous illustre l'utilisation des clauses else et finally.

```
try:
    num = int(input('Entrer un nombre: '))
except ValueError:
    print('Vous devez entrer un nombre')
else:
    print('Vous avez bien entré un nombre')
finally:
    print('Ce texte sera toujours imprimé')
```

Comme signalé précédemment, le bloc else est exécuté quand l'exception ne produit pas c'est-à-dire lorsque le bloc try fonctionne correctement. Quant au bloc finally, il sera toujours exécuté, même si un return, break ou continue intervient dans le code avant l'instruction finally. Exemple :

```
def myFunction():
    try:
        return 1
    except:
        pass
    finally:
        print(2)

myFunction()
```

1.10.6.3. Les exceptions déclenchées par l'utilisateur : l'instruction raise

L'instruction raise permet au programmeur de déclencher une exception. Par exemple :

```
raise NameError('MyError')
```

Cette instruction renvoie NameError: MyError

Une exception est déclenchée grâce au mot-clé raise. Il suffit alors d'indiquer le nom de l'exception levée, suivie éventuellement d'arguments. La syntaxe générale est la suivante: raise NomException(argument).

```
raise MyValueError('Ceci est mon erreur de test')
```

Si on souhaite déterminer si une erreur a été levée, mais sans la prendre en charge, il est possible de lever la même erreur dans le bloc except à l'aide de raise. Exemple :

```
try:
    raise MyException('Une exception')
except MyException:
    print('Il y  une exception ici')
    raise MyError
```

On peut aussi décider que le programme agira différemment selon l'erreur produite. Dans l'exemple suivant, le programme teste d'abord si l'erreur est de type ZeroDivisionError auquel cas il affiche le message division par zéro. Pour un autre type d'erreur, il regarde s'il y a d'autres instructions except qui s'y rapportent. S'il y en a une, il exécute les lignes qui la suivent, sinon, le programme s'arrête et déclenche une erreur

```
def inverseFunction(x):
    y = 1.0 / x
    return y
    try :
        print ((-2.1) ** 3.1)
        print (inverse (2))
        print (inverse (0))
    except ZeroDivisionError:
        print ("division par zéro")
    except Exception as exc:
        print ("erreur inprévue : ", exc.__class__)
        print ("message", exc)
```

Il est également possible d'associer une variable à l'erreur générée c'est-à-dire stocker l'erreur dans une variable, afin de la manipuler et d'obtenir des informations plus précises : la variable spécifiée dans la ligne except est alors associée à l'exception qui a été levée. Il est alors possible d'accéder aux attributs et aux arguments de l'exception concernée. Exemple :

```
try:
    raise Exception('Premier argument', 'Autre argument')
except Exception as error:
    print(type(error))
    print(error.args)
```

1.10.7. Documenter une fonction

Après avoir élaboré une fonction (surtout une fonction relativement long et complexe), il et fortement de la documenter afin de permettre à d'autres utilisateurs de se l'approprier rapidement. La documentation d'une fonction est généralement une chaîne de caractères qui fournit une générale de la fonction ainsi que les aides utiles. Cette description est généralement spécifiée après la déclaration du nom de la fonction avant la définition des autres blocs d'instructions. L'exemple ci-dessous illustre comment documenter une fonction et comment accéder à cette document en cas de besoin.

```
def volumeSphere():
    "Ce programme calcule le volume d'un cube. \n La fonction est définie \
avec un seul argument obligatoire r qui représente le rayon du cercle.\
Celui-ci peut prendre n'importe quelle valeur positive"
    r=float(input("Saisir le rayon de la sphère"))
    import math
    return (4 * math.pi * r**3)/3
```

Dans la définition de la fonction volumeSphere, la chaine de caractères ne joue aucun rôle fonctionnel dans le script : elle est traitée par Python comme un simple commentaire, mais qui est mémorisé comme une documentation interne sur la fonction. Cette document est stockée dans un attribut appelé __doc__. Pour afficher cette attribut, on fait:

```
print(volumeSphere.__doc__)
```

Si le script du programme est édité sous l'environnement IDLE (environnement d'édition de python), on voit apparaître cette chaîne documentaire dans une « bulle d'aide », chaque fois qu'on commence à écrire le nom de la fonction ainsi documentées.

1.11. Les modules python

Tout comme les dictionnaires sont des collections d'objets (listes, tuples, set, etc..), les modules sont des collections de fonctions qui permettent de réalisées des tâches apparentées. Par exemple, le module math, contient un certain nombre de fonctions mathématiques telles que sinus, cosinus, tangente, racine carrée, etc.

De nombreux modules sont déjà pré-installés dans la bibliothèque standard de python. Toutefois, pour réaliser certaines tâches spécifiques, on est souvent amené à installer des modules supplémentaires (ex : numpy, scipy, matplotlib, pandas, etc..)

Dans cette section, nous menons une discussion générale sur comment installer et utiliser un module python. Nous passerons en revue quelques modules incontournables de python en matière de Data Science.

1.11.1. Importation d'un module python

Dans la plupart des cas, avant d'utiliser un module, il faut d'abord importer dans la bibliothèque en utilisant le mot clé import. L'exemple ci-dessous illustre comment importer le module math.

```
import math
```

L'instruction import permet de rendre disponible tout le module (c'est-à-dire l'ensemble des fonctions qui le constitue). Pour voir l'ensemble des fonction qui forment un module, on utilise la fonction dir(). Exemple :

```
dir(math)
```

Pour avoir l'aide sur une fonction particulière dans le module math on utilise la fonction help(). Exemple :

```
help(math.gamma) # renvoie l'aide sur la fonction gamma du module math
```

Puisqu'une fonction contient une multitude de fonction, il est souvent inutile d'importer toutes ces fonctions en appelant. Si on a besoin de quelques fonctions seulement, on peut utiliser le mot clé from (voir exemple ci-dessous)

```
from math import sin # Importe la fonction sinus
from math import cos, sin, tan, pi # Importe respectivement les fonction cosinus, sinus, tangente et la valeur pi (3.14).

from math import * # Importe toutes les fonction associées à math (équivalent à import math)
```

Remarque : lorsque le nom du module est long, il peut être utile d'importer le module en utilisant un alias avec le mot clé « as » afin de lui attribuer un nom temporaire qui sera utilisé alternativement au nom initial dans le reste du programme. Exemple :

```
import matplotlib as plt
import requests  as rqs
```

On peut alors utiliser ces alias partout où on fait appel aux modules correspondant. Nous donnerons des exemples dans la suite de la section. Attention toutefois à bien choisir ces alias afin d'éviter des confusions avec les noms d'autres objets pythons ou les mots réservés.

1.11.2. Utilisation d'un module python

Les exemples ci-dessous montrent quelques utilisations des modules python après leur importation

1.11.2.1. Quelques utilisations de la fonction math

```
from math import *
v=16 # définit une variable v
x = math.sqrt (v)  # Renvoie la racine carrée de v
y = math.e(v) # Renvoie l'exponentiel de v
z = math.log(v) # Renvoie le logarithme népérien de v
```

1.11.2.2. Quelques exemples d'utilisation du module random

Le module random permet de générer des séquences de nombres aléatoires. C'est un module de la bibliothèquee standard de python tout comme le module math. Pour l'importer on fait :

```
import random
```

Pour avoir un aperçu sur l'ensemble des fonctions associées, on fait

```
dir(random)
```

Pour importer quelques fonction utiles, on a :

```
from random import random, randint, seed, uniform, randrange,sample, shuffle # Importe quelques fonctions utiles de random
```

La fonction random du module random permet de générer un nombre aléatoire compris entre 0 et 1. Exemple :

```
import random
x=random.random() # Renvoie un nombre aléatoire.
print(x)
```

La fonction randint() renvoie un nombre aléatoire entier sélectionner dans un intervalle défini par deux bornes a et b. Exemple :

```
import random
x=random.randint(5,17) # Renvoie un nombre aléatoire entier entre 5 et 17.
print(x)
```

La fonction uniform() est comme la fonction randint() à la différence que la fonction uniform génère un nombre aléatoire (réel) suivant une loi uniforme sur un intervalle donné. Exemple :

```
import random
x=random.uniform(5,17)  # Renvoie un nombre aléatoire uniforme réel entre 5 et 17.
print(x)
```

La particularité des fonctions de génération de nombres aléatoires c'est qu'elles renvoient toujours des nouvelles valeurs lorsqu'elles sont appelées. Alors que dans de nombreuses situations, on voudrait que le programme renvoie les mêmes valeurs afin de reproduire les résultats obtenus lors de l'exécution précédente. Pour cela, il faut fixer le paramètre seed(). Exemple :

```
for i in range(10) :
      import random
      random.seed(2016)
      x=random.random() # Renvoie 0.7379.
      print(x)
      x=random.randint(5,17) # Renvoie 12
      print(x)
      x=random.uniform(5,17) # Renvoie 16.78
      print(x)
```

Dans ce code, on a appliqué dix fois les fonction random(), randint() et uniform(). Mais puisque le seed est fixé à la même valeur (2016) alors les fonctions vont tirer les mêmes valeurs. D'où l'importance de l'utilisation de la fonction seed(). Pour annuler l'effet du seed et réinitialiser on utiliser la commande :

```
random.seed() # réinitialise le seed() à la valeur du système.
```

La fonction shuffle permute aléatoirement les éléments d'une liste. Exemple

```
import random
x = [1, 2, 3, 4] # Soit une liste initialement définie comme suit :
random.shuffle(x)  # Effectue une première permutation aléatoire des élements
print(x)
random.shuffle(x) # Effectue une seconde permutation des éléments
print(x)
```

NB : On pouvait aussi utiliser la fonction seed() afin de pouvoir reproduire les mêmes résultats après chaque permutation.

N'oublier pas également les fonctions randrange() et sample() qui permettent respectivement des tirages aléatoires (la première dans un intervalle avec un pas donné et la seconde un tirage aléatoire d'échantillon suivant le plan SI.

1.11.3. Définir et utiliser son propre module

On peut créer son propre module en rassemblant plusieurs fonctions dans un seul script et en l'enregistrant dans le répertoire courant avec l'extension .py. Le nom doit être un nom simple ne créant pas d'ambiguïté avec d'autres objets python. On peut choisir par exemple : myprogram.py

Une fois que le programme est enregistré dans le répertoire courant, il suffit d'importer le module comme un module classique et toutes ses fonctions (et variables) deviennent toute accessibles.

La première fois qu'un module est importé, Python crée un fichier avec une extension .pyc (ici myprogram.pyc) qui contient le bytecode (code précompilé) du module. L'appel du module se fait avec la commande:

```
import myprogram
```

Ensuite on peut utiliser les fonctions du module comme avec un module classique.

1.11.4. Procédure d'installation et d'utilisation des modules externes

Comme nous l'avons signalé un peu plus haut, de nombreux modules de python sont des modules externes qui doivent d'abord être installés afin de pouvoir effectuer certaines tâches. C'est le cas par exemples des modules de calcul scientifiques et numériques (scipy, numpy), les modules d'élaboration de graphiques (malplotlib) ou encore des modules e traiement de données, d'analyses statistiques ou de data science (pandas, Statmodels, scikit-learn, etcc).

Dans cette section, nous décrivons les étapes d'installations des modules externes ainsi que quelques astuces utiles pour une installation rapide.

En effet, pour installer un module externe python, il faut suivre les suivantes suivantes (Attention, cette méthode n'a été testée que sur python 3.4 installé sur un système Windows 7, 32bit)

1- Chercher le module sur Internet le module à télécharger et télécharger le fichier en extention whl (ou un dossier contenant les fichiers (dézipper si nécessaire). De nombreux fichier en whl peuvent être trouvés à l'adresse http://www.lfd.uci.edu/~gohlke/pythonlibs/

Télécharger le fichier et déposer le dans le dossier: "C:\Python34\Scripts" qui est le répertoire où est installé python 3.4

2- Lancer une page dans un éditeur de texte (Bloc-Note, Nordpad, Notepad, etcc). Enregistrer cette page quelque part, de préférence dans le répertoire "C:\Python34\Scripts" avec le nom avec l'extention .bat comme suit : "fichier_installation_modules.bat". Ce fichier étant créé, on va ajouter quelques lignes de commande qui installent automatiquement les modules qui seront indiqués. Pour ouvrir le fichier, faite clique-droit --> modifier. Le fichier avec l'extension .bat est un fichier exécutable. En faisant ouvrir, le programme s'exécute.

3-Après avoir ouvert le fichier (en mode modifier), ajouter les lignes de commande suivantes correspondant aux modules à installer:

```
cd C:\Python34\Scripts
pip install numpy-1.9.2+mkl-cp34-none-win32.whl
pip install scipy-0.15.1-cp34-none-win32.whl
pip install Pillow-2.8.1-cp34-none-win32.whl
pip install matplotlib-1.5.1-cp34-none-win32.whl
pip install rpy2-2.7.8-cp34-none-win32.whl
pip install pandas-0.17.0-cp34-none-win32.whl
pip install xlrd-0.9.4-py2.py3-none-any.whl
pip install xlrd-0.9.4-py2.py3-none-any.whl
pip install patsy
pip install requests-2.10.0-py2.py3-none-any.whl
pip install beautifulsoup4
pip install lxml-3.4.4-cp34-none-win32.whl
pip install html5lib-0.9999999-py2.py3-none-any.whl
pip install urllib3-1.16-py2.py3-none-any.whl
pip install scikit_learn-0.15.1-cp34-none-win32.whl
pip install nose-1.3.7-py3-none-any.whl
pip install statsmodels-0.6.1-cp34-none-win32.whl
pip install python_dateutil-2.5.3-py2.py3-none-any.whl
pip install mca
pip install nltk
pip install html2text
pip install openpyxl-2.4.1-py2.py3-none-any.whl
pip install XlsxWriter-0.9.3-py2.py3-none-any.whl
pause
```

4- Après avoir ajouté ces lignes de commande, on enregistre et on ferme le fichier. Ensuite, on clique-droit--> Exécuter ou Ouvrir ou double-cliquer. L'instruction pip install se lance alors et installe tous les modules les uns à la suite des autres. Il faut noter qu'on peut aussi installer les modules dans l'invite commande de windows.

Nb : Tous ces modules indiqués ici doivent être préalablement téléchargés et déposés dans le répertoire C:\Python34\Scripts. Le nombre de modules externes à installer dépend des besoins

de l'utilisateur. D'une manière générale l'installation de nouveaux modules se fait au fur et à mesure que les besoins de l'utilisateur augmentent. Cette longue liste de module n'est donc pas le fruit le fruit d'une installation d'un seul jour.

Conseil technique : Attention, pour un débutant, il est conseillé de ne pas avoir une version trop avancé de python sinon, vous risquez de ne pas pouvoir avoir tous les modules téléchargeables compatibles avec votre version. Il faut donc trouver un équilibre entre le fait d'être à jour dans les versions python et la compatibilité des fonctions que vous serez amenés à utiliser. Vous pouvez consulter par exemple le site http://www.lfd.uci.edu/~gohlke/pythonlibs/ pour analyser les modules disponibles pour quelles versions de python

Dans la liste des modules spécifiées ci-dessus, on remarque deux catégories : les fichiers avec l'extention whl et les fichiers sans extension. Les fichiers avec l'extention whl ont pour la plupart dans leur nom les mots « cp34-none-win32 » ou « py2.py3-none-any ». cp34-none-win32 signifie que le module est compatible avec la version 3.4 avec un système windows 32 bit. py2.py3-none-any signifie que le module est compatible avec python 2.x et 3.x et installable sur n'importe quel système.

Quant aux fichiers sans extensions, c'étaient au départ des fichiers sources (src) archives avec l'extension tar.gz que nous avons extrait et déposer comme un dossier dans le répertoire Scripts. Le module s'installe alors comme pour un fichier en whl. Notons toutefois que le nom des versions ne doit pas être spécifié par dans le programme d'installation. Par exemple si le dossier est nommé mca-1.0, on écrira simplement pip install mca.

Il faut aussi signaler qu'il existe une autre méthode d'installation plus simple des modules sous windows. Cela se fait à partir de l'invite commande de windows et le programme easy_install.exe présent dans le dossier Scripts. Par exemple, pour installer le module numpy, on tape la commande :

```
C:\Python34\Scripts\easy_install.exe numpy
```

Python fait lui-même la recherche du fichier d'installation (y compris sur internet) et installe le module.

Pour connaitre la version d'un module installé, on utilise la propriété __version__. Exemple : pour connaître la version de matplotlin, on fait comme suit :

```
Import matplotlib
matplotlib.__version__
```

1.11.5. Désinstaller un module

Pour désinstaller un package, on utilise la commande uninstall à la place install avec pip (sans les numéros des versions). Exemple : Pour désinstaller numpy, on fait :

pip uninstall numpy

1.12. Aperçu général sur les objets « classe »

Une classe est un objet qui sert à construire d'autres objets avec lesquels on associe des méthodes (ou fonction) propres à cet objet. Par exemples, la classe str() est utilisée pour créer des objets en chaînes de caractères. Plusieurs méthodes sont alors associées à cet objet comme par exemple les fonctions lower, upper, etc... Pour voir l'ensemble des méthodes associées à une objet on utilise la fonction dir(nomObjet).

Les objets peuvent posséder des attributs (variables associées aux objets) et des méthodes (fonctions associées aux objets)

L'exemple ci-dessous illustre la définition d'une classe.

On va créer une classe appelée Rectangle dans laquelle on va définir un ensemble de fonctions applicable à une rectangle.

```
class Rectangle:
    "ceci est la documentation de classe Rectangle"

### 1- initialisation d'un objet
    def __init__(self, long = 0.0, larg = 0.0): # définition des attributs d'un objet self avec des variables ayant des valeurs par défaut

# 2- Définition des attributs ayant pour valeur ces variables
        self.longueur = long # l'attribut longueur prend la valeur long
        self.largeur = larg # l'attribut largueur prend la valeur larg

# 3- definir une méthode (fonction) qui calcule la surface de self
    def calculSurface(self):
        print("surface = %.2f m2 "%(self.longueur * self.largeur))

# 4- definir une méthode( fonction) qui redéfinit le rectangle en carré
    def changeCarre(self, cote):
        self.longueur = cote
        self.largeur = cote
```

Ici, longueur et largeur sont des attributs alors que calculSurface() et changeCarre() sont des méthodes. Tous les attributs et toutes les méthodes se réfèrent toujours à self qui désigne l'objet lui-même. Attention, les méthodes prennent toujours au moins self comme argument

La classe rectangle ainsi définie peut être utilisée à d'autres fins (notamment calculer d'autres classes). Exemples :

```
rect1 = Rectangle() # crée un objet Rectangle avec les paramètres par défaut indiquée lors de la définition de la classe.
print(rect1.longueur, rect1.largeur)
rect1.calculSurface()  # applique la méthode qui calcule la surface (avec les valeurs par défaut ici)
rect1.changeCarre(30) #  Change le rectangle en carré de dimension 30
rect1.calculSurface() # applique de nouveau la méthode de calcul de surface avec les nouvelle dimensions
rect2 = Rectangle(2, 3) # Crée un objet Rectangle avec des paramètres indiqués
rect2.calculSurface() # Calcule la surface avec ces nouvelle dimensions
```

1.13. Utilisation du module pickle pour l'enregistrement et le chargement des objets python

Le module pickle permet d'enregistrer un objet python dans un fichier afin de pouvoir le recupérer ultérieurement. Le fichier résultant pourra être lu depuis n'importe quel système d'exploitation (à condition, que celui-ci prenne en charge Python). Les fonctios les plus utilisées sont pickler() et Unpicler() avec respectivement les méthodes dump(), load().

1.13.1. Utilisation de la fonction Pickler

Ouvrons un fichier nommé mesobjets sous le nom x comme suit:

```
import pickle
with open ('mesobjets', 'wb') as x : # le fichier est ouvert à la fois en mode w et b (écriture en binaire)
     pickle.Pickler(x).dump(nomobjet) # crée un objet pickler à partir de nomobjet pour enregistrer dans mesobjets x
```

On peut faire quelques remarques par rapport à la création de l'objet pickle. D'abord le fichier mesobjets n'a pas d'extension. On peut lui assigner une. Par exemple mesobjets.txt. Ensuite le fichier a été ouvert en mode d'écriture binaire wb. Ce fichier contiendra alors les écritures binaires. Ensuite tous les objets enregsitrés seront directement renvoyés dans mesobjets avec la fonction dump(). nomobjet peut être n'importe quel objet: liste, tuple, dictionnaire, etc...)

Exemple: soient les données sur des joueurs d'un jeu et leur score définie par un dictionnaire (ou bibliothèque) comme suit:

```
score = {" joueur 1": 5, " joueur 2": 35, " joueur 3": 20, " joueur 4": 2}
Enregistrons cet objet dans un pickle
import pickle
with open ('mesobjets.txt', 'wb') as x :
    pickle.Pickler(x).dump(score )
```

Pour ajouter d'autres objets on appelle de nouveau la méthode dump() avec les objets à enregistrer. Exemple

```
myobs1 = {'nom':'Jean', 'age':25, 'poids':70, 'taille':1.75}
myobs2 = {'nom':'Paul', 'age':30, 'poids':65, 'taille':1.80}
myobs3 = {'nom':'Pierre', 'age':35, 'poids':75, 'taille':1.65}
myobs4 = {'nom':'Baptiste', 'age':28, 'poids':80, 'taille':1.90}
with open ('mesobjets.txt', 'wb') as x :
    pickle.Pickler(x).dump(myobs1 )
    pickle.Pickler(x).dump(myobs2 )
    pickle.Pickler(x).dump(myobs3 )
    pickle.Pickler(x).dump(myobs4 )
```

En observant le contenu du fichier créé, on s'aperçoit que les données sont codées :

```
import os
os.system("cat mesobjets.txt")
```

1.13.2. Utilisation de la fonction Unpickler

La fonction unpickler() permet de récupérer les objets enregistrés par le pickler. Exemple : soit le pickler suivant

```
score = {" joueur 1": 5, " joueur 2": 35, " joueur 3": 20, " joueur 4": 2}
with open ('mesobjets.txt', 'wb') as x :
    pickle.Pickler(x).dump(score )
```

Pour récupérer les objets enregsitrés, on ouvre le fichier en mode 'r' et 'b' avec la fonction unpicker et la méthode load() à la place de dump()

```
with open ('mesobjets.txt', 'rb') as x :
     monobjet=pickle.Unpickler(x).load()
print(monobjet)
```

Attention à ne pas utiliser la fonction load() une fois de trop, sinon une erreur est générée. Il est à noter qu'il existe un module équivalent mais beaucoup plus rapide, le module cpickle.

Chapitre 2 : Etude des chaînes de caractères et des expressions régulières

2.1. Les chaînes de caractères

Les chaînes de caractères ne sont pas des objets en tant que tels dans un programme python. Ce sont plutôt des séquences de valeurs (tout comme les données de type numériques) qui servent à définir les objets proprement dits comme les listes, les tuples, etc…

A l'image des opérations mathématiques réalisables les données numériques (addition, multiplication, division, etc..), il existe également un ensemble d'opérations de traitements et de manipulation des chaînes de caractères. Cette section a pour but de passer en revues certaines de ces opérations les plus couramment rencontrées notamment dans les opérations de traitement de texte.

Avant d'introduire la section, il faut d'abord faire une remarque importante concernant la différence entre une valeur de type caractère et une valeur de type alphabétique. En effet, une valeur alphabétique est toujours une chaîne de caractère alors qu'une chaîne de caractère n'est pas nécessairement une valeur alphabétique. Sous python, une valeur caractère est toujours exprimée entre guillemets (simple, double ou triple) alors qu'une valeur numérique est exprimée sans guillemets. On distingue deux principaux types de données : les données numériques et les données en caractères. Il faut noter qu'une valeur numérique exprimée entre guillemets est automatiquement traitée comme une valeur caractère même si elle n'est pas alphabétique. Il est important de garder à l'esprit ces détails lors de la manipulation des séquences de valeurs sous python (voir exemple ci-dessous)

```
x= 12 # x est une variable numérique
y="12" #  y est une chaîne de caractère formée par une chiffres
z="mon texte" # z est une chaîne de caractères formée de valeurs alphabétiques
k="Ce stylo coûte 5 euros" # k est une chaîne de caractères formée de valeur alphanumérique
```

Nb : Noter aussi que dans une chaîne de caractères, l'espace (blank) est un caractère qui représente une valeur à part entière dans la chaîne. Il nécessite souvent des opérations de traitement spéciales, à défaut de quoi, il se comporte comme un élément de la chaîne au même titre que les autres caractères. Nous reviendrons plus amplement sur ces aspects dans la suite de cette section.

2.1.1. Définition d'une variable de type chaîne de caractères

Une variable chaîne de caractère se défini de manière classique par une assignation directe en utilisant le symbole de l'égalité (comme cela a été discutée dans la section sur la création de variable). La seule particularité des variables caractères par rapport aux variables numériques

est que leurs valeurs doivent être spécifiées entre guillemets lors de l'assignation. Les trois exemples ci-dessous sont des illustrations.

```
ch1 = 'Seriez-vous à la réunion de ce soir ?'
ch2 = '"Oui", répond-il'
ch3 = "D'accord, j'apprécierai bien"
ch4 = """Cette phrase est une longue chaîne de caractères contenant tous les types de guillemets: " ' « » mais aussi de nombreux caractères spéciaux"""
print(ch1)
print(ch2)
print(ch3)
print(ch4)
```

Ces quatre exemples montrent différentes utilisations des guillemets lors de la définition d'une variable en chaines de caractères.

Dans le premier exemple (ch1), on utilise des guillemets simples car il n'y a aucun inconvénient à cela compte tenu de la chaîne spécifiées. On pouvait aussi utiliser des guillemets doubles.

Dans le deuxième exemple (ch2), on utilise des guillemets simples car la chaînes spécifiée contient déjà des guillemets doubles comme valeur.

Dans le troisième exemple (ch3), on utilise les guillemets doubles car le texte contient des apostrophes qui sont en fait des guillemets simples.

Dans le quatrième exemple (ch4), on utilise les guillemets triples car le texte contient non seulement des guillemets simples, apostrophes et les guillemets doubles mais s'étend sur plusieurs lignes. Dans ces conditions, l'utilisation des guillemets triples s'impose.

2.1.2. Indiçage des chaînes de caractères (slicing)

Une chaîne de caractère est une séquence de valeurs ordonnées et indicées. Ce qui signifie qu'on peut accéder à chacun des éléments de la séquence en spécifiant son indice comme dans les liste. Les exemples ci-dessous sont des illustrations.

Soit la chaîne de caractères ch définie par :

```
ch="Christelle"
```

On peut faire les slicing suivant :

```
ch[0] # renvoie 'C'
ch[2] # renvoie 'r'
ch[-1] # renvoie 'e', le dernier élément de ch
ch[:6] # Renvoie 'Christ'
ch[6 :] # Renvoie 'elle'
ch[0:10:2] # renvoie 'Crsel'
```

Remarque : les éléments d'une chaîne de caractères ne sont pas définie en fonction des mots séparés par des espace mais par les caractères qui le constituent. En effet, même si la chaîne avait été constituée par une phrase de plusieurs mots, les indiçages se feront uniquement sur la base des caractères qui forment la longue chaînes. Exemple : soit la chaîne définie par :

```
ch="Ceci est une chaîne de plusieurs mots"
```

En faisant les indiçages sur cette chaîne, on a :

```
ch[1] # renvoie 'e'
ch[4] # renvoie ' ', l'espace vide qui sépare Ceci et est.
ch[0:4] # renvoie 'Ceci'
```

2.1.3. Déterminer la longueur d'une chaîne de caractères (nombre de caractères)

Pour déterminer la longueur d'une chaîne, on utilise la fonction len() comme pour une liste ou pour tout autre objet python. L'exemple ci-dessus est une illustration.

```
ch="Ceci est une chaîne de plusieurs mots"

print(len(ch)) # renvoie 37
```

La chaîne ch est constituée de 37 caractères (espace compris).

2.1.4. Addition de chaînes de caractères (concaténation)

L'addition de chaînes de caractères est la juxtaposition de chaînes de caractères les unes à la suite des autres pour une former une chaîne de caractères unique. Il s'agit de la concaténation.

Pour réaliser cette opération, on utilise le symbole + entre les noms des variables chaînes. Le résultat obtenu est ensuite assignée à une nouvelle variable. Exemples :

```
x = 'Un petit pas pour l'homme,'
y = 'un grand pas pour l'humanité'
z = a + b
print(z) # renvoie 'Un petit pas pour l'homme, un grand pas pour l'humanité'
```

Attention toutefois dans l'utilisation de l'opérateur de concaténation lorsqu'il s'agit d'associer une valeur numérique à une chaîne pour former une chaîne unique. Exemple

```
x= 'Le prix du stylo est'
y=5
z= 'euros'
```

On veut concaténer x, y et z de sorte à obtenir la phrase 'Le prix du stylo est 5 euros'.

Dans cette situation, on ne peut pas faire x+y+z car y est de type numérique. Il faut d'abord convertir celle-ci en utilisant la fonction str(). Ainsi on a :

```
ch=x+str(y)+z
print(ch) # renvoie Le prix du stylo est5euros
```

On voit que est 5 euros est sans espace, on peut alors modifier l'opération de concaténation en insérant des espace entre les valeurs. Ainsi, on a :

```
ch=x + ' '+ str(y) + ' '+ z
print(ch) # renvoie Le prix du stylo est 5 euros
```

Ce résultat pouvait aussi être obtenu par une expression directe telle que :

```
ch='Le prix du stylo est'+ ' '+ '5' + ' '+ 'euros'
print(ch) # renvoie Le prix du stylo est 5 euros
```

2.1.5. Modifier la casse d'une chaîne en majuscule, minuscule ou capital : fonctions upper(), lower() et capitalize()

Modifier la casse d'une chaîne, c'est de convertir l'ensemble de la chaîne en majuscule, en minuscule tout autre casse comme la casse capital (chaîne commençant par une lettre majuscule).

Pour réaliser ces trois opérations, on utilise respectivement les fonctions upper(), lower(), capitalize(). Les trois exemples ci-dessous sont des illustrations.

Soit la chaîne définie par :

```
ch="Ceci est une chaîne de plusieurs mots"

print(ch.lower()) # renvoie : ceci est une chaîne de plusieurs mots
print(ch.upper()) # renvoie : CECI EST UNE CHAÎNE DE PLUSIEURS MOTS
print(ch.capitialize()) # renvoie : Ceci est une chaîne de plusieurs mots
```

Remarque: Penser à utiliser souvent les fonctions lower() et upper() dans certaines tâches de recherche de matching des mots afin d'harmoniser les casses dans les deux chaines à matcher pour la recherche.

2.1.6. Rechercher un caractère (ou un motif) dans une chaîne et renvoyer son indice : fonction find()

La fonction find() permet de parcourir une chaîne de caractères et renvoyer l'indice (position) du premier caractère qui forme la chaîne (ou le motif) recherché. Exemple : Soit la chaîne ch définie comme suit :

```
ch="Sa chambre est située au 9ième étage tout au fond du couloir"
```

On peut avoir quelques applications de la fonction find() comme suit :

```
ch.find("étage") # Renvoie 31 (qui correspond à l'indice du caractère de début "é"
ch.find("au") # Renvoie 22 (indice de "a" de la première occurrence du motif "au".
ch.find("u") # renvoie 18, première occurrence du motif "u".
```

Nb : Une chaîne de caractères est toujours traitée comme un « tout ». Aucune distinction n'est faite entre les espaces et les autres caractères. Ce qui signifie que le comptage des indices renvoyés inclut aussi ceux des espaces. Par ailleurs, un motif est toujours recherché selon qu'il soit au milieu d'un mot donné ou séparé des autres par des espaces. Cela transparait dans les exemples ci-dessus. En effet, dans le premier exemple, le motif "étage" est séparé des autres caractères par des espaces. Mais en renvoyant l'indice du caractère "é", tous les caractères situés avant sont d'abord parcourus (y compris les espaces). Ensuite, dans l'exemple 2, le motif "au" se répète deux fois dans la chaine, alors seul l'indice de la première occurrence dans la chaîne initiale est renvoyé par défaut. Toutefois, il est possible de modifier ce comportement en indiquant restreignant l'intervalle dans lequel la rechercher du motif doit être effectué. Pour cela, on utilise l'option start et end. Pour avoir l'aide complète sur la fonction find(). Il faut simplement taper la commande help(ch.find).

Pour ce qui concerne le troisième exemple, il s'agit de renvoyer l'indice de la première occurrence du motif "u". On peut constater que ce motif plusieurs fois dans la chaîne et est généralement localisé à l'intérieur des mots ("située", "tout" et "couloir"). Mais comme nous venons de le signaler, peu importe que le motif recherche soit localisé à l'intérieur d'autres ensemble de caractères ou séparé des autres par des espaces, la fonction find() tout comme la plupart des autres fonctions de traitement de chaines de caractères considère la chaîne initiale comme un bloc unique de caractères. Le comptage des indices est toujours effectué à partir du début de la chaîne, quelque celle-ci commence ou pas par un espace.

Notons aussi que lorsque le motif recherche n'est pas trouvé, la fonction find() renvoie la valeur -1. Ce qui peut être utilisé comme une valeur booléenne pour construire d'autres instructions notamment les instructions conditionnelles.

Rappelons enfin que la fonction find() est l'équivalent de la fonction index() lorsqu'il s'agit des listes. En effet, nous avons montré que dans le cas des listes pour renvoyer l'indice d'un élément en se basant sur sa valeur, on utilise la fonction index(). Voir section sur les listes.

2.1.7. Rechercher un caractère (un motif) dans une chaîne et remplacer par un autre : fonction replace()

La fonction replace() permet de rechercher un motif dans une chaîne de caractères et le remplacer par une autre valeur. Exemple : soit la chaîne ch définie par :

```
ch="Sa chambre est située au 9ième étage tout au fond du couloir"
```

On souhaite remplacer "Sa chambre" par "Son bureau" et "située" par "situé". Alors, on fait :

```
ch.replace("Sa chambre", "Son bureau")
ch.replace("située" , "situé")
print(ch) # Renvoie "Son bureau est situé au 9ième étage tout au fond
du couloir"
```

2.1.8. Compter le nombre d'occurrence d'un motif dans une chaîne : fonction count()

Tout comme dans le cas des listes, pour compter le nombre d'occurrences d'un motif dans une chaine, on utilise la fonction count(). Exemple : soit la fonction ch définie par

```
ch="Son bureau est situé au 9ième étage tout au fond du couloir"
```

On peut utiliser la fonction count() comme suit :

```
ch.count("au") # renvoie 3
ch.count("i") # renvoie 3
ch.count("tout") # renvoie 1
```

2.1.9. Découper une chaîne de caractères en liste : fonction list() ou fonction split()

On peut distinguer deux principaux cas de décomposition d'une chaîne en liste : cas où les espaces sont traités comme des caractères et cas où les espaces sont traités comme des séparateurs. Lorsque les espaces présents dans une chaînes doivent être traités comme des valeurs à part entière, pour décomposer la chaine initiale en liste, on utilise la fonction list(). En revanche lorsque les espaces doivent être traités comme des séparateurs de caractères, alors on utiliser la fonction split(). Ces deux fonctions sont détaillées ci-dessous

2.1.9.1. Cas où les espaces sont traités comme des valeurs (list())

Dans le cas où les espaces doivent être récupérés comme des valeurs, pour décomposer une chaîne de caractères en liste, on utilise la fonction list(). La fonction list() récupère chaque caractère de la chaîne (espace compris) et le stocke dans une liste comme un élément à part entière. Exemple : soit la chaine de caractères définie comme suit :

```
ch="Ceci est une chaîne avec des espaces"
```

On décompose cette chaîne en liste comme suit :

```
chList=list(ch)
print(chList)#renvoie: ['C', 'e', 'c', 'i', ' ', 'e', 's', 't', ' ', 'u', 'n', 'e', ' ', 'c', 'h', 'a', 'î', 'n', 'e', ' ', 'a', 'v', 'e', 'c', ' ', 'd', 'e', 's', ' ', 'e', 's', 'p', 'a', 'c', 'e', 's']
```

On remarque que la fonction list() procède au découpage de la chaine sur la base des caractères individuels (ce qui n'est pas le cas pour la fonction split()) et la liste finalement obtenue contient les espaces ' ' comme éléments au même titre que les autres caractères.

On pouvait aussi utiliser une boucle « for » sur chaque caractère de la chaîne en le récupérant et le stockant dans une liste initialement vide. Exemple :

```
chList=[] # Création liste vide
for i in ch:
    chList.append(i)
print(chList)#renvoie: ['C', 'e', 'c', 'i', ' ', 'e', 's', 't', ' ', 'u', 'n', 'e', ' ', 'c', 'h', 'a', 'î', 'n', 'e', ' ', 'a', 'v', 'e', 'c', ' ', 'd', 'e', 's', ' ', 'e', 's', 'p', 'a', 'c', 'e', 's']
```

2.1.9.2. Cas où les espaces sont traités comme des séparateurs (split())

Dans le cas où les espaces doivent être traités comme des séparateurs de caractères et non des valeurs, pour décomposer une chaîne de caractères en liste, on utilise la fonction split(). La fonction split() récupère chaque caractère (ou ensemble de caractères) de la chaîne (en ignorant les espaces compris) et le stocke dans une liste comme un élément. Exemple : soit la chaine définie comme suit :

```
ch="Ceci est une chaîne avec des espaces"
```

On décompose cette chaîne en liste comme suit :

```
chList=ch.split()
print(chList)#renvoie: ['Ceci', 'est', 'une', 'chaîne', 'avec', 'des', 'espaces']
```

On remarque que la fonction split() découpe la chaîne non pas sur la base des caractères individuels mais sur la base des ensembles de caractères séparés par des espaces. Le découpage est donc réalisé en fonction des mots qui constituent la chaîne initiale. La liste finale obtenue ne contient pas d'espace. Ceux-ci servent à délimiter les éléments qui doivent former la liste.

Remarque : Par défaut, la fonction split() considère l'espace comme séparateur des éléments qui doivent former la liste. Mais on peut modifier le comportement en indiquant un séparateur personnalisé (comme - , ; , $, #) en spécifiant l'option sep=. Exemple : Soit la chaine suivante :

```
ch="Ceci-est-une- -chaîne-avec-des-tirets-et un espace"
```

Cette chaîne peut se décompose en liste comme suit :

```
chList=ch.split("-")
print(chList)#renvoie: ['Ceci', 'est', 'une', ' ', 'chaîne', 'avec', 'des', 'tirets', 'et un espace']
```

En observant cette liste, on peut faire plusieurs remarques. D'abord, l'espace est considéré comme un élément à part entière de la liste lorsque celui-ci est délimiter de part et d'autre par le séparateur spécifié (ex : "- - "). Ensuite, tout ensemble de caractère qui n'inclut pas le symbole du séparateur sera considéré comme un élément de la liste en tant que tel (ex : 'et un espace').

Rappel : Cette section étudie comment transformer une chaîne en liste. Rappelons aussi que qu'il est possible de rassembler une liste pour former une chaîne de caractères unique (cela a été discuté dans les sections sur les listes). Il faut utiliser la fonction join() avec ou sans séparateur. Voir l'exemple de rappel ci-dessous :

```
mylist = ["A", "T", "G", "A", "T"]
ch1= " ".join(mylist) # jointure avec espace comme séparateur
ch2= "-".join(mylist) # jointure avec tiret comme séparateur
ch3= "".join(mylist) # jointure avec sans espace
```

2.1.10. Les opérations logiques et booléennes sur les chaines de caractères

Tout comme pour les variables, il existe un certain nombre de fonctions permettant d'effectuer des opérations logiques et booléennes sur les chaines de caractères. Dans la plupart des cas, il s'agit de tester si une condition logique est vérifiée sur une chaine auquel cas, des instructions peuvent alors être définies. Cette section a pour but de présenter brièvement certaines de ces fonctions logiques.

2.1.10.1. La fonction startwith()

Il permet de vérifier si une chaine de caractère commence par un motif donné. Ex :

```
mytext = "this is string example....wow!!!";
print (mytext.startswith( 'this' )) # renvoie : True
```

Attention : Par défaut, la fonction startwith() considère toute la chaîne indiquée et vérifie si le motif spécifié se trouve en début de caractère. Toutefois, il est possible de modifier ce comportement en indiquant un intervalle de test (voir exemples ci-dessous)

```
print (mytext.startswith( 'is', 2, 4 )) # Teste si le motif se trouve entre le 3ième et le 5ième caractère. Renvoie : True
print (mytext.startswith( 'this', 2, 4 )) # renvoie : False
```

2.1.10.2. la fonction endwith()

A l'inverse de la fonction startwith(), la fonction endwith() teste si une caractère ou une portion de chaine de caractères finit par un motif donné. Exemple :

```
mytext = "this is string example....wow!!!"
print (mytext.endwith( 'this' )) # renvoie : False
print (mytext.endwith( 'wow!!!' )) # renvoie : True
print (mytext.endwith( 'is', 2, 4 )) # Renvoie : True
print (mytext.endwith( 'this', 2, 6 )) # renvoie : False
```

2.1.10.3. La fonction in

La fonction in permet de vérifier si un motif donné se trouve dans une chaine. Exemple :

```
mytext = "this is string example....wow!!!"
"this" in mytext # Renvoie True
"that" in mytext # Renvoie False
```

2.1.10.4. La fonction islower()

La fonction islower() permet de vérifier si l'ensemble des caractères d'une chaine est en minuscule. Exemples :

```
mytext1 = "this is string example....wow!!!"
mytext1.islower() # renvoie True
mytext2 = "This is string example....Wow!!!"
mytext2.islower() # renvoie False
```

2.1.10.5. La fonction isupper()

A l'instar de la fonction islower(), la fonction isupper() permet de vérifier si l'ensemble des caractères d'une chaine est en majuscule. Exemples :

```
mytext1 = "this is string example....wow!!!"
mytext1.isupper() # renvoie False
mytext2 = "THIS IS STRING EXAMPLE....WOW!!!"
mytext2.isupper() # renvoie True
```

2.1.10.6. La fonction istitle()

la fonction istitle() permet de vérifier si chaque caractère (ou ensemble de caractères) commence par une majuscule. Le format title (titre) est une phase dans laquelle chaque mot commence par une lettre majuscule et le reste en minuscule. Cette convention permet par exemple d'identifier les titres ou les sous-titre dans un ensemble de texte. L'exemple ci-dessous montre l'utilisation de la fonction istitle ():

```
mytext1 = "this is string example....wow!!!"
mytext1.istitle() # renvoie False
mytext2 = "This Is String Example....Wow!!!"
mytext2.istitle() # renvoie True
mytext3 = "This is String Example....Wow!!!"
mytext3.istitle() # renvoie False
mytext4 = "THIS IS STRING EXAMPLE....WOW!!!"
mytext4.istitle() # renvoie False
```

2.1.10.7. La fonction isalpha()

La fonction isalpha() permet de vérifier si l'ensemble des caractères d'une chaine est en lettre alphabétique (de a à z ou A à Z). Pour que la fonction renvoie True, la chaine ne doit pas contenir de chiffres, de caractères spéciaux, ni d'espace Exemples :

```
mytext1 = "this is string example....wow!!!"
mytext1.isalpha() # renvoie False
mytext2 = "thisisstringexamplewow"
mytext2.isalpha() # renvoie True
mytext3 = "thisisstringexamplewow250"
mytext3.isalpha() # renvoie False
mytext4 = "this-is-string-example-wow"
mytext4.isalpha() # renvoie False
```

2.1.10.8. La fonction isalnum()

La fonction isalnum() permet de vérifier si l'ensemble des caractères d'une chaine est en *alphanumérique* (de a à z ou A à Z, de 0 à 9). Pour que la fonction renvoie True, la chaine ne doit pas contenir de caractères spéciaux, ni d'espace. Exemples :

```
mytext1 = "this is string example....wow!!!"
mytext1.isalnum() # renvoie False
mytext2 = "thisisstringexamplewow"
mytext2.isalnum() # renvoie True
mytext3 = "thisisstringexamplewow250"
mytext3.isalnum() # renvoie True
mytext4 = "thisisstringexamplewow250_"
mytext4.isalnum() # renvoie False
```

2.1.10.9. La fonction isdigit()

La fonction isdigit() permet de vérifier si l'ensemble des caractères d'une chaine est en *chiffre* (allant de 0 à 9). Pour que la fonction renvoie True, la chaine ne doit pas contenir, ni de caractères alphabétiques, de caractères spéciaux, ni d'espace. Exemples :

```
mytext1 = "this is string example....wow!!!"
mytext1.isdigit() # renvoie False
mytext2 = "250406879"
mytext2.isdigit() # renvoie True
mytext3 = 250406879
mytext3.isdigit() # renvoie une erreur car mytext3 n'est pas de type caractère (mais numérique)
mytext4 = "thisisstringexamplewow250"
mytext3.isdigit() # renvoie False
```

Nb : Consulter la page http://www.tutorialspoint.com/python/python_strings.htm pour plus de fonctions concernant les opérateurs booléens sur les chaines.

2.1.11. Formatage de valeurs à l'intérieur d'une chaine de caractères

Il arrive très souvent que l'on veuille récupérer la valeur d'une variable et l'intégrer à une chaine de caractères pour former une nouvelle chaine de caractères utilisable à d'autres fins. Par exemple, pour l'affichage (avec la fonction print()) des valeurs d'une variable en plein milieu d'une chaine de texte formant par exemple un message, on a souvent besoin de formater d'abord la valeur de la variable avant de pouvoir le concaténer avec les autres caractères pour former le message. Car, comme nous l'avons montré précédemment, il n'est pas possible de concaténer la valeur d'une variable numérique avec celle d'une variable caractères sans convertir celle-ci d'abord en chaine de caractère en utilisant par exemple la fonction str(). Cette opération de conversion se ramène très souvent à un formatage. Le formatage d'une valeur sous python est une opération très spéciale qui mérite une longue explication. Le but de cette section est de donner un aperçu général sur les méthodes de formatage de valeurs.

Pour introduire la notion de formatage de valeur, nous allons partir d'un exemple concrèt. Soit la variable prix définie comme suit

```
prix=2
```

Nous voulons afficher un message de genre "`Le prix du stylo est de 2 euros`". Dans ce message, 2 correspond à la valeur de la variable prix définie ci-dessus. Mais prix étant une variable, on souhaite que le message aussi se modifie lorsque le prix change. Par exemple lorsque prix=5 alors on aura le message "`Le prix du stylo est de 5 euros`". Le but de l'opération de l'opération de formatage de faire afficher la valeur d'une variable à l'intérieur d'une chaine de caractères. Pour cela, on dispose de trois principales méthodes.

- l'utilisation de l'opérateur de concaténation "+ "

- l'utilisation de l'opérateur de formatage "%" %

- l'utilisation de la fonction format "{}".format()

Nous passerons en revue chacune de ces trois méthodes avec des exemples d'application à l'appui.

2.1.11.1. Formatage avec l'opérateur de concaténation "+"

Soit les variables nom, age poids, taille définies comme suit :

```
nom="Julien"
age =25
poids=75
taille=1.70
```

Nous souhaitons afficher le message suivant : `"Julien est âgé de 25 ans. Il pèse 75 kilos et mesure 1.70 m."`

Pour constituer une telle chaîne, on utilise la concaténation suivante :

```
ch=nom + " " + "est âgé de" + " " + str(age)+ " " + "ans." + " " + "Il pèse" + " " + str(poids) + " " + "kilos et mesure" + " " + str(taille)+ " " + "m."
print(ch) # renvoie Julien est âgé de 25 ans. Il pèse 75 kilos et mesure 1.7 m.
```

On peut faire un certain nombre de remarque sur cette opération de formatage.

D'abord, les variables numériques (age, poids, taille) ont été toutes d'abord converties en variables caractère en utilisant la fonction str(). Car il n'est pas possible de concaténer une valeur numérique et une chaine sans cette transformation préalable. Cette opération de conversion n'est toutefois pas nécessaire pour la variable nom qui est déjà en caractères. Ensuite les éléments qui doivent constituer la phrase à afficher sont associées en utilisant l'opérateur de concaténation +. Remarquons également l'ajout des caractères espaces afin d'aérer la phrase sinon le texte final sera tout collé et illisible.

2.1.11.2. Utilisation de l'opérateur de formatage "%" %

L'exemple ci-dessous montre les différentes utilisations de l'opérateur de formatage "%" % pour afficher la valeur d'une variable à l'intérieur d'une chaîne.

Soit les variables nom, age poids, taille définies comme suit :

```
nom="Julien"
age =25
poids=75
taille=1.70
```

Nous souhaitons afficher le message suivant : "Julien est âgé de 25 ans. Il pèse 75 kilos et mesure 1.70 m." en utilisant l'opérateur de formatage "%" %. Alors on a :

```
ch="%s est âgé de %i ans. Il pèse %.0f kilos et mesure %.2f m." %(nom,
age, poids, taille)
print(ch) # renvoie Julien est âgé de 25 ans. Il pèse 75 kilos et
mesure 1.70 m.
```

Dans cette méthode de formatage, l'ensemble du texte est spécifiée dans un guillemet unique. Ensuite, on met l'opérateur de formatage % là où la valeur doit apparaître tout en indiquant le type de la variable. Par exemple %s indique que la valeur à faire à ce niveau est de type string (s). %i signifie une valeur entière (integer), %.0f signifie float avec 0 chiffre après la virgule et %.2f signifie float avec 2 chiffres après la virgule (on distingue plusieurs autres types de format : ex : %d qui signifie valeur décimal).

Enfin, pour le deuxième opérateur % spécifié à l'extérieur des guillemets, on doit y indiquer les noms des variables par ordre d'apparition dans la chaine. La variable nom est la première à faire apparaitre. Elle se situe au niveau de %s. la variable age se situera au niveau de %i, etc... Les noms des variables sont indiqués entre parenthèses lorsqu'il y en a plusieurs. Mais cela n'est pas nécessaire lorsqu'il s'agit d'une seule variable.

2.1.11.3. Utilisation de la fonction format()

La fonction format() est une spécification améliorée de l'opérateur de formatage "%". L'exemple ci-dessous illustre son utilisation.

Soit les variables nom, age poids, taille définies comme suit :

```
nom="Julien"
age =25
poids=75
taille=1.70
```

Nous souhaitons afficher le message suivant : "Julien est âgé de 25 ans. Il pèse 75 kilos et mesure 1.70 m." en utilisant la fonction format(). Alors on a :

```
ch="{0} est âgé de {1} ans. Il pèse {2} kilos et mesure {3}
m.".format(nom, age, poids, taille)
print(ch) # renvoie Julien est âgé de 25 ans. Il pèse 75 kilos et
mesure 1.70 m.
```

Dans cette méthode de formatage, l'opérateur %type de la précédente méthode est remplacé par des accolades en y indiquant l'indice correspondant à la liste des variables indiquée à l'extérieur des guillemets. Les accolades indiquent là où les valeurs doivent être affichées et les indices les variables correspondant. En effet l'indice 0 correspond à la variable nom dans la liste de variables indiquée dans format(nom, age, poids, taille). {1} correspond à la variable age, ainsi de suite.

Nb : On pouvait aussi remplacer les indices 0, 1, ..., par des noms de valeurs comme suit :

```
ch="{vNom} est âgé de {vAge} ans. Il pèse {vPoids} kilos et mesure
{vTaille} m.".format(vNom=nom, vAge=age, vPoids=poids, vTaille=taille)
print(ch) # renvoie Julien est âgé de 25 ans. Il pèse 75 kilos et
mesure 1.70 m.
```

2.1.11.4. Formatage de valeurs indicées et création de variables indicées

Il arrive très souvent que dans l'utilisation d'une boucle « for » ou « while », on génère des valeurs ou des variables indicées. Pour le cas des valeurs indicées, on veut simplement les afficher et pour le cas des variables indicées, on veut les assigner des valeurs. Les deux exemples ci-dessous illustrent la mise en œuvre de ces deux types de cas.

Formatage de valeurs indicées

On souhaite par exemple afficher les chaînes de caractères indicées comme suit : value1, value2,..., value10. On peut simplement utiliser les trois méthodes précédemment discutées à l'intérieur d'une boucle. Voir exemple ci-dessous :

```
listVal=list(range(1,11)) # Création d'un liste de valeur allant de 1
à 10
# Création de valeurs formattée
for v in listVal :
        v1= "value%s"%v  # méthode1: opérateur %
        print(v1)
        v2= "value{0}".format(v)  # méthode2: opérateur format()
        print(v2)
        v3= "value"+str(v)  # méthode3: opérateur +str()
        print(v3)
```

Les valeurs générées ci-dessus peuvent ensuite être assignées à des variables pour être utilisées dans le reste du programme.

Cependant, il arrive qu'on veuille générer plutôt des variables auxquelles on va assigner des valeurs quelconques. Pour cela, on combine les méthodes présentées avec la fonction globals (voir ci-dessous).

Formatage de variables indicées

On veut créer 10 variables nommée var1, var2,..., var10 et les assigner une valeur quelconque (ex:100). Alors on a:

```
listVal=list(range(1,11)) # Création d'un liste de valeur allant de 1
à 10
# Création de valeurs formattée
for v in listVal :
        globals()["var%s"%v]=100  # Utilisation de l'opérateur %
```

Les 10 variables sont créées. On peut simplement afficher le contenu de n'importe quel variable en faisant print(). Ex :

```
print(var5)
```

NB : Il faut remarquer que ces créations de variables indicées sont faites à l'image de la création des valeurs indicées avec l'opérateur % en ajoutant juste la fonction global()[ ]. Il est possible aussi d'utiliser la fonction locals().

Pour plus de détails sur le traitement des chaines de caractères consulter la page : http://www.tutorialspoint.com/python/python_strings.htm

2.1.12. Utilisation de l'opérateur antislash \ : mise en forme de la chaîne et traitement des caractères spéciaux

L'opérateur antislash joue plusieurs rôles dans un programme python. D'abord, il permet de traiter des caractères spéciaux tels que les guillemets et les apostrophes, lorsque ceux-ci se trouvent à l'intérieur d'une chaîne (pour éviter que python les considère comme des débuts ou fins de chaînes). L'opérateur antislash permet alors d'indiquer que ces caractères doivent être traités comme faisant partie des caractères et pas comme des délimiteurs de chaînes. Dans ce cas, l'opérateur antislash joue le rôle d'opérateur d'échappement.

Une autre fonction de l'opérateur antislash est de réaliser un certain nombre de mise en forme de la chaîne de caractère (création de saut de ligne, saut d'espace, insertion de tabulation, etc). Dans ce cas l'opérateur antislash joue le rôle de mise en forme de la chaîne.

Les exemples ci-dessus illustrent quelques utilisations de l'opérateur antislash.

Soit la chaine de caractères ch définie comme suit :

```
ch = 'Ceci est une longue chaîne qui déborde sur une ligne et s\'étend sur plusieurs. Mais cela n\'est pas un problème \
car on peut utiliser l'opérateur antislash \ pour indiquer que la ligne continue. \n Toutefois, on peut insérer une nouvelle ligne'

print(ch)
```

Cette chaine montre différente utilisation de l'opérateur \.

D'abord, une simple spécification de \ en fin de ligne signifie que la ligne continue et qu'il ne s'agit pas d'une nouvelle ligne.

D'abord, une simple spécification de \ en milieu de ligne n'a aucune signification particulière. Le symbole \ est alors traité comme un élément de la chaine.

La spécification \ devant les apostrophes permet de traiter ceux-ci comme des caractères et non des délimiteurs de chaînes. C'est le cas pour les mots "s'étend" et "n'est". En effet, puisque la chaine ch est définie à l'intérieur des guillemets simples, il y aura confusions lorsque des délimiteurs si l'opérateur \ n'est pas utilisé pour échapper les apostrophes.

La spécification \n indique à python d'insérer une nouvelle ligne et d'afficher ce qui suit sur cette ligne. C'est pourquoi en faisant print(ch), la portion " `Toutefois, on peut insérer une nouvelle ligne`" apparait sur une nouvelle ligne alors que la portion " `car on peut utiliser l'opérateur antislash \ pour indiquer que la ligne continue.` " est affichée sur la première ligne.

L'opérateur antislash peut donc s'avérer utile dans de nombreuses situations. Les spécifications ci-dessous illustrent d'autres utilisations de l'opérateur à l'intérieur d'une chaine.

\ Ignore le saut de ligne (lorsque placé en fin de ligne)
\\ Echappe un antislash dans la chaîne
\' Echappe un apostrophe (utile si la chaine est définie à l'intérieur des guillemets simples)
\" Echappe un guillemet échappe un apostrophe (utile si la chaine est définie à l'intérieur des guillemets doubles)
\a Insère une sonnerie (bip)
\b Provoque un retour arrière
\f Insère un saut de page
\n Insère un saut de ligne
\r Provoque un retour en début de ligne
\t Insère une tabulation horizontale
\v Insère une tabulation verticale
\N{nom} Insère un caractère sous forme de code Unicode nommé
\uhhhh Insère un caractère sous forme de code Unicode bits
\Uhhhhhhhh Insère un caractère sous forme de code Unicode bits
\ooo Insère un caractère sous forme de code octal
\xhh Insère un caractère sous forme de code hexadécimal

2.2. Etudes des expressions régulières (regex ou re)

2.2.1. Définition et rôle des expressions régulières

Dans la section consacrée à l'étude des chaînes de caractères au chapitre 1, nous avons montré que la fonction str() était bien capable de retrouver des patterns (motifs) simples et d'effectuer des opérations de traitement nécessaires. Toutefois, lorsque la définition du pattern devient plus complexe, il faut faire appel aux expressions régulières. En effet, les expressions régulières sont des extensions de la fonction str() qui permettent de réaliser des opérations plus complexes de définition de motifs que ne permettent des fonction str(). C'est un outil très puissant qui permet de vérifier si le contenu d'une variable a la forme (pattern) de ce que l'on attend. Par exemple si on récupère un numéro de téléphone, on s'attend à ce que la variable soit composée de nombres (compris entre 0 et 9) et d'espaces (ou de tiret) mais rien de plus.

Les expressions régulières permettent non seulement de vous avertir d'un caractère non désiré mais également de supprimer/modifier tous ceux qui ne sont pas désirables.

2.2.2. Les opérateurs d’expression régulière

Le module python permettant de traiter les expressions régulières est re. Ce module doit d’abord être importé comme suit :

```
import re
```

Les symboles les plus souvent utilisés pour définir une expression régulières sont : . ^ $ * + ? { } [ ] \ | ( )

Chacun de ces signes correspond à une opération bien précise qui permet de définir un pattern. Voici ci-dessous les significations:

. Le point correspond à n'importe quel caractère. Quand on le met à un endroit dans un pattern, cela veut dire que tout caractère peut être à ce niveau quelque soit le nombre de caractères. Ce caractère représente donc n’importe quel caractère. Exemple : l’expression A.G peut correspondre par exemples à ATG, AtG, A4G, mais aussi à A-G ou à A G.

^ Indique un commencement de segment (quand il est à l'extérieur des crochet) mais signifie aussi "contraire de" (quand il est à l'intérieur des crochet). En pratique, il désigne le début d'une nouvelle ligne puisq'il indique de renvoyer la proposition si il n'y a aucun caractère devant (Attention l'espace est un caractère). Exemple : l’expression ^ATG peut matcher la chaîne de caractères ATGCGT mais pas à la chaîne CCATGTT.

$ Fin de segment. C'est à dire tous les mots ou expressions finissant par la suite de caractères indiquées. En pratique $ désigne une fin de ligne, puisqu'aucun caractère n'est censé être présent après ce caractères (y compris les espaces !). Exemple : l’expression ATG$ peut matcher la chaîne de caractères TGCATG mais pas avec la chaîne CCATGTT.

[xy] Une liste de segment possibble. Exemple [abc] équivaut à : a, b ou c (NB: le ou inclut le et). [ABC] le caractère A ou B ou C (un seul caractère). Exemple : l’expression T[ABC]G correspond à TAG, TBG ou TCG, mais pas à TG.

[A-Z] n’importe quelle lettre majuscule. Exemple : l’expression C[A-Z]T correspond à CAT, CBT, CCT...

[a-z] n’importe quelle lettre minuscule

[0-9] n’importe quel chiffre

[A-Za-z0-9_] n’importe quel caractère alphanumérique

[^AB] n’importe quel caractère sauf A et B. Exemple : l’expression CG[^AB]T correspond à CG9T, CGCT... mais pas à CGAT ni à CGBT.

(x|y) Indique un choix multiple; équivaut à "x" OU "y". (CG|TT) chaînes de caractères CG ou TT. Exemple : l'expression A(CG|TT)C correspond à ACGC ou ATTC. NB: Quand les parenthèses sont spécifiées, elles s'appliquent à l'ensemble formé par les caractères et non individuellement

\d le segment est composé uniquement de chiffre. Ce qui peut donc s'écrire comme [0-9]. Tous les caractères rencontrés sont des chiffres (peu importe leur nombre)

\D le segment n'est pas composé de chiffres, ce qui équivaut à [^0-9].

\s Un espace, ce qui équivaut à [\t\n\r\f\v] qui sont les séparateur classiques \t, \n, \r, \f ou \v (voir la section UTILISATION DE L'ANTISLASH \ .

\S Pas d'espace, ce qui équivaut à [^ \t\n\r\f\v].

\w Présence alphanumérique, ce qui équivaut à [a-zA-Z0-9_].

\W Pas de présence alphanumérique [^a-zA-Z0-9_].

\ Est un caractère d'échappement (permettant de continuer un ligne, ou écrire une apostrophe, etc...). Exemple : l'expression \+ désigne le caractère + sans autre signification particulière. L'expression A\.G correspond à A.G et non pas à A suivi de n'importe quel caractère, suivi de G.

? Définit 0 à 1 fois le caractère précédent ou l'expression entre parenthèses. Exemple: GR(.)?S . Résultats possibles: GRS, GROS, GRIS, GRAS. L'expression A(CG)?T correspond à AT ou ACGT.

+ Définit 1 à n fois le caractère précédent ou l'expression entre parenthèses précédente. Exemple: GR(.)+S . Résultats possibles: GROS, GRIS, GRAS. L'expression A(CG)+T correspond à ACGT, ACGCGT... mais pas à AT.

* Définit 0 à plusieurs fois le caractère précédent ou l'expression entre parenthèses précédente. Exemple: GR(.)*S. Résultats possibles: GRS,GROO,GRIIIS,GROlivierS. L'expression A(CG)*T correspond à AT, ACGT, ACGCGT...

{ } permet d'imposer le nombre d'occurences d'un caractères ou une expression. Les différentes variantes sont :

{n} n fois le caractère précédent ou l'expression entre parenthèses précédente

{n,m} n à m fois le caractère précédent ou l'expression entre parenthèses précédente

{n,} au moins n fois le caractère précédent ou l'expression entre parenthèses précédente

{,m} au plus m fois le caractère précédent ou l'expression entre parenthèses précédente

Exemples: A{2} : on attend à ce que la lettre A (en majuscule) se répète 2 fois consécutivement ; BRA{,9} : on attend à ce que le segment BRA s répète de 0 à 9 fois consécutives ; BA{1,9} : on attend à ce que le segment BA se répète de 1 à 9 fois consécutives ; VO{1,} : on attend à ce que le segment VO soit présent au mois une fois.

2.2.3. Exemples d'application des opérateurs d'expressions régulières

Avant d'approfondir l'utilisation des regex , on va d'abord faire quelques exercices pour la définition des patterns. Pour cela, on va définir des patterns et les tester sur quelques textes. Nous allons nous servir de la fonction match pour faire nos tests.

Exercice 1 : Soit le pattern suivant: "GR(.)?S" Vérifions si le texte "GRIS" correspond à ce pattern. Pour faire ce test avec le module re, on utilise la fonction match comme suit :

```
import re
print (re.match(r"GR(.)?S", "GRISS"))   # s'il ya match, l'objet est
crée et la fonction renvoie la valeur None
```

Exercice 2 : Donner la signification de chacun des patterns ci-dessous et dire si le texte en question est matché.

pattern	texte
GR(.)+S	GRIS
GR(.)?S	GRS
GRA(.)?S	GRAS
GAS(.)?	GRAS
GR(A)?S	GRAS
GR(A)?S	GRS
M(.)+N	MAISON
M(.)+(O)+N	MAISON
M(.)+([a-z])+N	MAISON
M(.)+([A-Z])+N	MAISON
^!	!MAISON!
!MAISON	!MAISON!
^!MAISO!$	!MAISON!
^!MAISON!$	!MAISON!
^!M(.)+!$	!MAISON!
([0-9])	03 88 00 00 00
^0[0-9]([.-/]?[0-9]{2}){4} 03 88 00 00 00	
^0[0-9]([.-/]?[0-9]{2}){4}	03/88/00/00/00
^0[0-9]([.-/]?[0-9]{2}){4}	03_88_00_00_00

Résolution:

NB : On peut vérifier les corrections en utilisant la fonction match() comme dans l'exercice 1.

GR(.)+S signifie 1 ou plusieurs caractères de n'importe quel type entre GR et S. Le texte GRIS est matché.

GR(.)?S signifie 0 ou 1 caractère au plus de n'importe quel type entre GR et S. Le texte GRS est matché.

GRA(.)?S signifie 0 ou 1 caractère au plus de n'importe quel type entre GR et S. Le texte GRAS est matché.

GAS(.)? signifie 0 ou 1 caractère au plus de n'importe quel type à partir de GAS. Le texte GRAS n'est pas matché

GR(A)?S signifie 0 ou 1 fois la lettre A entre GR et S. Le texte GRAS est matché.

GR(A)?S signifie 0 ou 1 fois la lettre A entre GR et S. Le texte GRS est matché

M(.)+N signifie 1 ou plusieurs caractères de n'importe quel type entre M et N. MAISON est matché.

M(.)+(O)+N signifie 1 ou plusieurs caractères de n'importe quel type entre M et O qui lui même se repète 1 ou plusieurs fois avant N. MAISON est matché

M(.)+([a-z])+N signifie 1 ou plusieurs caractères de n'importe quel type entre M et toute lettre alphabétique minuscule (repété 1 ou plusieurs fois) avant N. MAISON n'est pas matché

M(.)+([A-Z])+N signifie 1 ou plusieurs caractères de n'importe quel type entre M et toute lettre alphabétique majuscule(repété 1 ou plusieurs fois) avant N. MAISON est matché

^! Signifie tout mot commençant par !. Le mot !MAISON! est matché

!MAISON Ici le pattern ne contient pas d'opération. Donc le pattern est tout simplement !MAISON. !MAISON! est macthé car le pattern est toujours une partie du texte

^!MAISO!$ Signifie tout caractère commençant par !MAISO! et se terminant par ce meme mot. Donc !MAISON! pas matché

^!MAISON!$ Signifie tout caractère commençant par !MAISON! et se terminant par ce même mot. Donc !MAISON! est matché

^!M(.)+!$ Signifie tout caractère commençant par(1 ou plusieurs caractères de n'importe quel type entre !M et !) et se terminant par ce mot. !MAISON! est doncmatché.

([0-9]) signifie tout nombre entre 0 et 9 se repetant autant de fois que nécessaire. Alors 03 88 00 00 00 est matché.

^0[0-9]([.-/]?[0-9]{2}){4} signifie..... le nombre 03 88 00 00 00 est matché

^0[0-9]([.-/]?[0-9]{2}){4} 03/88/00/00/00 (matché)

^0[0-9]([.-/]?[0-9]{2}){4} 03_88_00_00_00 (pas matché)

2.2.4. Construire une expression : la fonction compile()

2.2.4.1. Le rôle de la fonction compile()

Au delà de l'intérêt de vérifier si un texte indiqué matche ou pas avec le pattern, l'intérêt majeur des expressions régulières réside dans le traitement de texte ou dans le texte mining. Souvent on est amené à rechercher dans un texte un certain pattern et faire le traitement nécessaire.

Parmi les opérations les plus souvent rencontrées, on a la recherche d'un texte ou la recherche et remplacement d'un texte. Lorsque le texte à rechercher ou à remplacer repond à un certaine logique lexicale, on peut alors définir une expresion régulière.

Toutefois, pour un pattern donné, lorsque l'expression régulière qui le traduit est relativement simple et que cette expression ne sera pas utilisé dans le programme de manière réccurente, on peut se contenter d'une simple définition (définition ponctuelle).

Exemple:

```
x=re.findall("([0-9]+)", "Bonjour 111 Aurevoir 222") # recherche dans le texte "Bonjour 111 Aurevoir 222", le pattern([0-9]+
print(x) # Il s'agit ici d'une recherche simple.
Pour faire une recherche et un remplacement, on fait:
x= re.sub(r"Bienvenue chez (?P<chezqui>\w+) ! Tu as (?P<age>\d+) ans ?", r"\g<chezqui> a \g<age> ans", "Bienvenue chez olivier ! Tu as 33 ans")
print(x) # renvoie: olivier a 33 ans
```

Dans ces deux exemples ci-dessus, on utilise les fonctionss find.all() et sub() qui sont des fonctions associées aux regex(on reviendra sur ces fonctions plus bas).

Les deux exemples ci-dessus sont des utilisations ponctuelles et simples des re dans le programme. Mais lorsque l'utilisation d'un pattern est complexe et repététive dans le programe, il faut définir un objet qui enregistre le pattern en question. Pour cela, on utilise la fonction compile()

2.2.4.2. Utilisation de la fonction compile() sans directives

Définissons un pattern(motif) qui identifie dans un texte toutes les chaines commençant par b ou B suivies de n'importe quelles lettres de a-z ou A-Z et compilons cette expression pour l'enregister dans un obejt que nous allons appeler au cours du programme plus tard. On a :

```
w=re.compile(r"[bB][a-zA-Z]*")
```

Le pattern w ainsi défini peut être ensuite être appliqué à n'importe quelle chaine de caractères pour renvoyer une liste

Exemple 1

```
resultat1 =w.findall("pi vaut 3.14 et e vaut 2.72")
print(resultat1) #il n'y a aucun aucun mot qui commence par b ou B alors la liste resultat1 est vide.
```

Exemple 2

```
phrase = "Zorro zozotte contrairement à Bernardo"
resultat2 =w.findall(phrase)
print(resultat2) #la liste resultat2 est contient Bernardo.
```

2.2.4.3. Utilisation de la fonction compile() avec directives

La syntaxe générale de l'utilisation de la fonction compile est la suivante :

cpattern = re.compile(pattern, directives)

Les directives se présentent sous forme d'un entier : chaque directive est une puissance de deux, et le paramètre directives est la somme de directives particulières. Voici- ci-dessous quelques directives

Directive	abrégé	valeur	description
IGNORECASE	I	2	ignorer la casse ; fonctionne avec les lettres accentuées
LOCALE	L	4	définir comme "lettre" ce que la langue locale définit comme tel dans la variable système.;
MULTILINE	M	8	considérer le texte comme décomposé en lignes (le caractère \n est le début de chaque ligne)
DOTALL	S	16	considérer le saut de ligne comme un caractère ordinaire.
VERBOSE	X	64	permettre d'écrire des commentaires dans les patrons.
ASCII	A	256	permettre de travailler en ASCII

NB : On peut écrire indifféremment re.IGNORECASE+re.MULTILINE ou re.I+re.M

2.2.5. Quelques fonctions associées aux expressions régulières

Tout comme pour la fonction str() dans le cas du traitement direct des chaînes de caractères, il y a aussi plusieurs fonctions associées à re. Ci-dessous la description de quelques-unes.

2.2.5.1. La fonction search()

La fonction head(permet de rechercher le motif (pattern) au sein d'une chaîne de caractères avec une syntaxe de la forme search(motif, chaine). Si motif existe dans chaine, Python renvoie une instance MatchObject.

Exemple 1:

```
animaux = "girafe tigre singe"
x=re.search('tigre', animaux)
print(x) # donne _sre.SRE_Match object; span=(7, 12), match='tigre'>.
Ce qui signifie que le mot a été bien trouvé
```

Exemple2: test

```
if re.search('tigre', animaux):

    print "OK"
```

Exemple 3: recherche chaines commençant par b ou B

```
z=re.compile(r"[bB][a-zA-Z]*") # chaines commençant par b ou B suivies
de n'importe quelles lettres de a-z ou A-Z
phrase = "Zorro zozotte contrairement à Bernardo"
x=z.search(phrase) # on cherche la premiere occurence de l'objet z
dans phrase
print(x)   "   donne   _sre.SRE_Match   object;   span=(30,   38),
match='Bernardo'
```

Exemple 4 : On veut trouver toutes les occurences du mot "tache" dans le texte suivant :

```
phrase = "Qui mange avec une moustache se tache."
z = re.compile(r"tache")
x=z.search(phrase) # on cherche la premiere occurence de l'objet z
dans phrase
print(x)
```

Exemple 5 : On veut trouver toutes les postions des occurences du mot "tache" . On ajoute la fonction span

```
z = re.compile(r"tache")
z.search(phrase).span() # retourne 23,28
```

Le problème avec la fonction search est que l'on trouve la première occurence, même si elle est dans un autre mot. Pour éviter ce problème on peut utiliser \b qui est un délimiteur de début ou de fin de mot:

```
z = re.compile(r"\btache\b")
x=z.search(phrase).span() # retourne 32,37
#   \B sert à s'assurer que l'on est pas un début ou une fin de mot (
c'est la situation négative de \b) :
texte = "Une moustache on s'y attache, même si ça tache."
```

```
p = re.compile(r"\w*\Btache\w*|\w*tache\B\w*")
x=p.findall(texte)
print(x) # renvoie ['moustache', 'attache']
```

2.2.5.2. La fonction match()

Dans le module re, la fonction match()fonctionne comme la fonction search(). La différence est qu'elle renvoie une instance MatchObject seulement lorsque l'expression régulière correspond (match) au début de la chaîne (à partir du premier caractère).

Exemple 1:

```
z=re.compile(r"[bB][a-zA-Z]*") # chaines commençant par b ou B suivies
de n'importe quelles lettres de a-z ou A-Z
phrase = "Zorro zozotte contrairement à Bernardo"
x=z.match("Bernardo") # renvoie None si aucun prefixe n'est
reconnaissable par l'expression
```

Exemple 2:

```
animaux = "girafe tigre singe"
re.match('tigre', animaux)
```

Utilisation comparée des fonctions match er search

La recherche (search()) consiste à parcourir le texte depuis le début ; si le pattern n'est pas identifiable dans le texte, la fonction retourne None. Sinon, elle retourne une instance de l'objet MatchObject, appartenant lui aussi au module _sre.

La comparaison (match()) consiste à identifier le début du texte au pattern. Elle retourne None si aucune comparaison n'est possible. Sinon elle retourne une instance MatchObject. * l'objet : MatchObject. Cet objet peut être interrogé par ses méthodes. Pour l'instant, les méthodes start() (premier caractère reconnu) et stop() (position après le dernier caractère reconnu)

Exemple:

```
texte = "Un IDE ou \"environnement de développement\" est un logiciel
\
constitué d'outils qui facilitent l'écriture et les tests dans un \
langage défini, voire plusieurs.\
\nCet IDE comporte en général un éditeur avec coloration syntaxique,\
un système de gestion de fichiers (sauvegarde/chargement),\
un compilateur, un exécuteur de programme, un système d'aide en
ligne,\
des indicateurs de syntaxe etc. \
\nLe plus connu est peut être Éclipse."
```

Recherche du pattern dans le texte ; résultat affiché : 158 165

```
pattern = "Cet IDE"
cpattern = re.compile(pattern)
resultat = cpattern.search(texte)
if resultat :
    print(resultat.start(), resultat.end())
else:
    print (resultat)
```

Comparaison du pattern au texte ; résultat affiché : None

```
resultat = cpattern.match(texte)
if resultat :
    print(resultat.start(), resultat.end())
else :
    print(resultat)
```

Comparaison du pattern au texte ; résultat affiché : 0 6

```
pattern = "Un IDE"
cpattern = re.compile (pattern)
resultat = cpattern.match(texte)
if resultat :
    print(resultat.start(), resultat.end())
else :
    print (resultat)
```

Recherche en ignorant la casse ; résultat affiché : 413 420

```
pattern = "éclipse"
cpattern = re.compile (pattern, re.IGNORECASE)
resultat = cpattern.search(texte)
if resultat :
    print(resultat.start(), resultat.end())
else:
    print(resultat)
```

2.2.5.3. La fonction findall()

Cette fonction recherche et retourne tous les éléments qui correspondent dans une liste de chaînes.

Exemple 1 :

```
regex = re.compile('[0-9]+\.[0-9]+')
resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
print(resultat) # donne ['3.14', '2.72']
regex = re.compile('([0-9]+)\.([0-9]+)')
resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
```

```
print(resultat)  # donne [('3', '14'), ('2', '72')]
```

Exemple 2 : soit la chaine suivante

```
phrase = "Zorro zozotte contrairement à Bernardo"
Définissons deux re telles que:
a=re.compile(r"[\w]+") # mots d'au moins une lettre de \w toutes ie. toutes les lettres (y compris accents)
b=re.compile(r"[bB][a-zA-Z]*") # chaines commençant par b ou B suivies de n'importe quelles lettres de a-z ou A-Z
```

Application de la fonction findall

```
x=a.findall(phrase) # on cherche tous les a dans phrase et renvoie une liste de ceux-ci
y=b.findall(phrase)
```

Exemple 3: on dispose d'une liste de noms de localités et l'on se propose de rechercher celles dont le nom de termine par ville. On ne distingue pas les majuscules et les minuscules dans la recherche. Les villes en questions sont indiquées comme suit:

Ablon

Acqueville

Agy

Aigner-Ville

Airan

Amayé-sur-Orne

Amblie

Amfreville

Angervillers

Angoville

Résolution :

```
On peut constituer le texte à partir de cette liste de villes comme suit :
texte ="Ablon\nAcqueville\nAgy\nAigner-Ville\nAiran\nAmayé-sur-Orne\
\nAmblie\nAmfreville\nAngervillers\nAngoville\nArganchy\nArgences\
\nArromanches-les-Bains\nAsnelles\nAsnières-Surville"
Définition du motif :
pattern = ".*ville$"
```

```
cpattern = re.compile (pattern)
resultat = cpattern.findall(texte)
print (resultat, "\n")
cpattern = re.compile (pattern, re.MULTILINE)
resultat = cpattern.findall(texte)
print (resultat, "\n")
cpattern = re.compile (pattern, re.MULTILINE+re.IGNORECASE)
resultat = cpattern.findall(texte)
print (resultat, "\n")
```

On a: ".*ville$" : une suite de caractères, suivie de ville, suivie de la fin de texte. Les fins de chaîne ne sont pas incluses. Seule la dernière ligne peut être prise en compte si on ne met pas de directives !

2.2.5.4. La fonction fonction sub()

La fonction sub() permet d'effectuer des remplacements assez puissants. Par défaut la fonction sub(chaine1,chaine2) remplace toutes les occurrences trouvées par l'expression régulière dans chaine2 par chaine1. Si on souhaite ne remplacer que les n premières occurrences, on utilise l'argument count=n : sub(chaine1,chaine2, n).

Exemple 1 :

```
regex = re.compile('([0-9]+)\.([0-9]+)')
regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72") # donne 'pi vaut quelque chose et e vaut quelque chose'
regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72", count=1) # donne 'pi vaut quelque chose et e vaut 2.72'
```

Exemple2

```
texte = "Retourne le texte avec ses remplacements. La \
valeur remplacement est une chaîne (qui peut être obtenue \
par application d'une fonction, appelée à chaque \
remplacement). count donne le maximum de remplacements ; \
0 est la valeur par défaut et signifie que tous les remplacements\
possibles doivent être effectués."
pattern= "\. " # antislash de protection du point
remplacement ="\n\n"
cpattern = re.compile(pattern)
nouveau = cpattern.sub(remplacement, texte)
print (nouveau)
print ("\n**********************************\n")
retTuple = cpattern.subn (remplacement, texte)
print (retTuple)
```

2.2.5.5. La fonction findinter()

Cette fonction agit comme findall() mais renvoie un itérateur.

Exemple :

b=re.compile(r"[zZ][a-zA-Z]")

phrase = "Zorro zozotte contrairement à Bernardo"

h=b.finditer(phrase) # comme findall mais renvoie un itérateur

2.2.5.6. La fonction group()

Il arrive souvent qu'on veuille récupérer la valeur d'un mot ou d'un ensemble de mots qui matche une partie de l'expression régulière. Pour cela les expressions régulières offrent la possibilité de créer des groupes de motifs à conserver.

Exemple : Conserver le suffixe des verbes se terminant par aient.

```
 s = 'ils allaient et bavardaient en cheminant'
rgx = r'(\w*)aient\b' # Le groupe que l'on veut conserver est indiqué entre ()
for m in re.finditer(rgx,s) :
    print m.group(0), m.group(1)
```

Les résultats sont: allaient all ; bavardaient bavard

groupe(0) est l'expression entière, groupe(1) correspond au premier groupe demandé (ici un seul).

Exemple 2 :

```
regex = re.compile('([0-9]+)\.([0-9]+)')
resultat = regex.search("pi vaut 3.14")
resultat.group(0) # donne 3.14
resultat.group(1) # donne 3
resultat.group(2) # donne 14
resultat.start() # donne 8
resultat.end() # donne 12
```

Dans cet exemple, on recherche un nombre composé de plusieurs chiffres [0-9]+ \. (le point a une signification comme métacaractère, donc il faut l'échapper avec \ pour qu'il ait une signification de point), suivi d'un nombre à plusieurs chiffres [0-9]+. Les parenthèses dans l'expression régulière permettent de créer des groupes qui seront récupérés ultérieurement par la fonction group(). La totalité de la correspondance est donné par group(0), le premier élément entre parenthèse est donné par group(1) et le second par group(2). Les fonctions start() et end() donnent respectivement la position de début et de fin de la zone qui correspond à l'expression régulière. Notez que la fonction search() ne renvoie que la première zone qui correspond à l'expression régulière, même s'il en existe plusieurs :

```
resultat = regex.search("pi vaut 3.14 et e vaut 2.72")
resultat.group(0) # donne 3.14
```

NB : Certains caractères spéciaux dans nos expressions régulières sont modélisés par l'anti-slash \. Vous savez sans doute que Python représente d'autres caractères avec ce symbole. Si vous écrivez dans une chaîne \n, Python effectuera un saut de ligne ! Pour symboliser les caractères spéciaux dans les expressions régulières, il est nécessaire d'échapper l'anti-slash en le faisant précéder d'un autre anti-slash. Cela veut dire que pour écrire le caractère spécial \w, vous allez devoir écrire \\w. C'est assez peu pratique et parfois gênant pour la lisibilité. C'est pourquoi on peut utiliser le mot clé r avant le délimiteur qui ouvre notre chaîne, tous les caractères anti-slash qu'elle contient sont échappés.

Exemple:

```
r' \n'
r' \texte'
```

Ci-dessous quelques opérateurs d'échappement de caractères spéciaux dans les regex :

\ symbole d'échappement

\e séquence de contrôle escape

\f saut de page

\n fin de ligne

\r retour-chariot

\t tabulation horizontale

\v tabulation verticale

\d classe des nombres entiers

\s classe des caractères d'espacement

\w classe des caractères alphanumériques

\b délimiteurs de début ou de fin de mot

\D négation de la classe \d

\S négation de la classe \s

\W négation de la classe \w

\B négation de la classe \b

2.2.6. Etude de quelques expressions régulières couramment rencontrées

(cas fournis par google support: https://support.google.com/a/answer/1371417?hl=fr#Match-Exact-Phrase-Only)

2.2.6.1. Cas 1 : Correspondance avec une expression exacte uniquement

Exemple: Rechercher une correspondance avec l'expression "stock tips".

```
###Possibilité de définition 1 :
(\W|^)stock\stips(\W|$)
##Possibilité de définition  2 :
(\W|^)stock\s{0,3}tips(\W|$)
##Possibilité de définition  3 :
(\W|^)stock\s{0,3}tip(s){0,1}(\W|$)
```

Remarques cas 1

\W désigne tout caractère qui n'est pas alphanumérique(lettre, chiffre ou un trait de soulignement). Ce métacaractère exclut la possibilité d'existence des caractères figurant avant ou après la proposition.

^ désigne le début d'une nouvelle ligne. Ce métacaractère indique à l'expression régulière de renvoyer la proposition si elle apparaît au début d'une ligne, c'est-à-dire sans caractère devant.

\s désigne un caractère d'espacement

Dans la possibilité 2, \s et {0,3} indique que zéro à trois espaces peuvent figurer entre les mots stock et tips.

$ désigne la fin d'une ligne. Ce métacaractère indique à l'expression régulière de renvoyer la proposition si elle apparaît à la fin d'une ligne, c'est-à-dire sans caractère derrière.

Dans la possibilité 3, (s) correspond à la lettre s et {0,1} indique que la lettre peut apparaître zéro ou une fois après le mot "tip".

Ainsi, l'expression régulière fait référence à stock tip et à stock tips. Vous pouvez également utiliser le caractère ? à la place de {0,1}.

2.2.6.2. Cas 2 :Correspondance avec un mot ou une expression figurant dans une liste

Exemple: Rechercher une correspondance avec tout mot ou toute expression parmi les mots suivants : baloney , darn, drat, fooey, gosh darnit,heck

```
###Possibilité de définition:
```

```
(\W|^)(baloney|darn|drat|fooey|gosh\sdarnit|heck)(\W|$)
```

Remarques cas 2

(...) regroupe tous les mots, de sorte que la classe de caractère \W s'applique à tous les mots figurant à l'intérieur des parenthèses.

\W désigne tout caractère qui n'est pas une lettre, un chiffre ou un trait de soulignement. Ce métacaractère évite que l'expression régulière ne prenne en compte des caractères figurant avant ou après les mots ou les propositions de la liste.

^ désigne le début d'une nouvelle ligne. Ce métacaractère indique à l'expression régulière de renvoyer le mot s'il apparaît au début d'une ligne, c'est-à-dire sans caractère devant.

$ désigne la fin d'une ligne. Ce métacaractère indique à l'expression régulière de renvoyer le mot s'il apparaît à la fin d'une ligne, c'est-à-dire sans caractère derrière.

| désigne un "ou", de sorte que l'expression régulière corresponde à n'importe quel mot de la liste.

\s désigne un espace. Utilisez cette classe de caractères pour séparer les mots d'une proposition.

2.2.6.3. Cas 3 :Correspondance avec un mot présentant des variantes orthographiques ou des caractères spéciaux

Exemple: Rechercher une correspondance avec le mot "viagra" et certaines autres orthographes pouvant être utilisés par les spammeurs telles que :

vi@gra , v1agra, v1@gra, v!@gr@

```
###Possibilité de définition:
v[i!1][a@]gr[a@]
```

Remarques cas 3:

Le métacaractère \W n'a pas été inclus dans l'expression de sorte que d'autres caractères puissent apparaître avant ou après chaque variante orthographique de viagra.

Par exemple, l'expression régulière correspond quand même à viagra dans les exemples suivants : viagra!! ou ***viagra***

[i!1] désigne les caractères i, ! ou 1 qui peuvent apparaître en deuxième position dans le mot.

2.2.6.4. Cas 4 : Correspondance avec une adresse e-mail sur un domaine spécifique

Exemple: Rechercher une correspondance dans toutes les adresses e-mail des domaines yahoo.com, hotmail.com et gmail.com.

```
###Possibilité de définition:
(\W|^)[\w.+\-]{0,25}@(yahoo|hotmail|gmail)\.com(\W|$)
```

Remarques cas 4

\W désigne tout caractère qui n'est pas une lettre, un chiffre ou un trait de soulignement. Ce métacaractère évite que l'expression régulière ne prenne en compte des caractères figurant avant ou après l'adresse e-mail.

^ désigne le début d'une nouvelle ligne. Ce métacaractère indique à l'expression régulière de renvoyer l'adresse si elle apparaît au début d'une ligne, c'est-à-dire sans caractère devant.

$ désigne la fin d'une ligne. Ce métacaractère indique à l'expression régulière de renvoyer l'adresse si elle apparaît à la fin d'une ligne, c'est-à-dire sans caractère derrière.

[\w.\-] renvoie tout caractère de mot (a-z, A-Z, 0-9 ou un trait de soulignement), un point, un signe plus ou un tiret.

Ce sont les caractères valides les plus couramment utilisés dans la première partie d'une adresse e-mail. Notez que \- (désignant un tiret) doit figurer à la fin de la liste de caractères incluse entre crochets. Sachez qu'il n'est pas nécessaire d'utiliser le caractère d'échappement \ pour le point situé entre crochets. Il ne signifie pas ici n'importe quel caractère. La barre oblique inversée \ figurant avant le tiret et le point (du .com) indique que le tiret et le point ne sont pas des caractères spéciaux appartenant à la syntaxe de l'expression régulière. Ils doivent être renvoyé quoi qu'il en soit. {0,25} indique que 0 à 25 caractères de la série précédente peuvent apparaître avant le signe @. Le paramètre de messagerie relatif à la conformité du contenu autorise une expression régulière à renvoyer jusqu'à 25 caractères pour chaque série de caractères.

Les parenthèses (...) regroupent les domaines et la barre verticale | qui les sépare signifie "ou".

2.2.6.5. Cas 5 :Correspondance avec une adresse IP comprise dans une plage

Exemple: Rechercher une correspondance avec toute adresse IP comprise dans la plage allant de 192.168.1.0 à 192.168.1.255.

```
###Possibilité de définition 1 :
192\.168\.1\.
###Possibilité de définition 2 :
192\.168\.1\.\d{1,3}
```

Remarques cas 5

La barre oblique inversée \ figurant avant chacun des points est un "caractère d'échappement" : elle indique que les points ne sont pas des caractères spéciaux appartenant à la syntaxe de l'expression régulière.

Dans l'exemple 1, aucun caractère ne vient après le dernier point, de sorte que l'expression régulière représente toute adresse IP commençant par 192.168.1., quel que soit le nombre qui suit.

Dans l'exemple 2, \d désigne tout chiffre compris entre 0 et 9, et {1,3} indique qu'un à trois chiffres peuvent figurer après le dernier point.

Dans ce cas, l'expression régulière représente toute adresse IP complète, commençant par 192.168.1.. Sachez que cette expression peut également renvoyer des adresses IP incorrectes, telles que 192.168.1.999.

2.2.6.6. Cas 6 : Correspondance avec un format alphanumérique

Exemple: Rechercher une correspondance dans les numéros de bons de commande de votre entreprise. Ils peuvent se présenter sous des formats différents, tels que :

PO nn-nnnnn ,PO-nn-nnnn , PO# nn nnnn ,PO#nn-nnnn, PO nnnnnn où le n sont des chiffres. ex: PO-25-2456

```
###Possibilité de définition:
(\W|^)PO[#\-]{0,1}\s{0,1}\d{2}[\s-]{0,1}\d{4}(\W|$)
```

Remarques cas 6

\W désigne tout caractère qui n'est pas une lettre, un chiffre ou un trait de soulignement. Ce métacaractère évite que l'expression régulière ne prenne en compte des caractères figurant avant ou après le numéro.

^ désigne le début d'une nouvelle ligne. Ce métacaractère indique à l'expression régulière de renvoyer le numéro s'il apparaît au début d'une ligne, c'est-à-dire sans caractère devant.

$ désigne la fin d'une ligne. Ce métacaractère indique à l'expression régulière de renvoyer le numéro s'il apparaît à la fin d'une ligne, c'est-à-dire sans caractère derrière.

[#\-] représente un signe dièse ou un tiret figurant après les lettres PO et {0,1} indique que ces caractères peuvent apparaître zéro ou une fois.

Notez que \- (désignant un tiret) doit figurer à la fin de la liste de caractères incluse entre crochets.

\s représente un espace et {0,1} indique qu'il peut apparaître zéro ou une fois.

\d correspond à tout chiffre compris entre 0 et 9, et {2} indique que deux chiffres exactement doivent figurer à cette position du numéro.

2.2.7. Exercices de synthèse sur l'utilisation des regex

Exemple 1: créer une expression qui reconnait une adresse mail valide

Résolution:

Admettons qu'une adresse Email a la forme générale suivante : XXXXXXX@XXXXX.COM

Commençons par rechercher XXXXXXXX@, cela peut se traduire par ^[a-z0-9._-]+@

Testons d'abord cela:

```
import re
string = "olivier@mailtest.com"
regexp = r"(^[a-z0-9._-]+@)"
if re.match(regexp, string) is not None:
    print ("TRUE")
else:
    print ("FALSE")
print (re.search(regexp, string).groups()) # Renvoie olivier@.
```

On voit qu'il y a match entre ce premier pattern et la première partie du complexe. On va maintenant ajouter la deuxième partir du format de l'adresse email. On a:

```
import re
string = "olivier@mailtest.com"
regexp = r"(^[a-z0-9._-]+@[a-z0-9._-]+\.[(com|fr)]+)"
if re.match(regexp, string) is not None:
    print ("TRUE")
else:
    print ("FALSE")
print (re.search(regexp, string).groups())
```

Exemple 2: Vérifier si un numéro de téléphone (français) est correct.

Résolution: les numéros français, commençent par 0 ou +33 et possédant 9 chiffres sans compter le 0 ou +33. Ainsi, voici une écriture possible de celle-ci :

```
myNumRegex=compile(r'(0|\+33)[1-9]( *[0-9]{2}){4}')
```

Exemple 3: Détecter les liens et les rendre cliquable

Soit le texte suivant:

```
mytexte ="""Bonjour et au revoir ! Je m'appelle Moussa KEITA Doe, j'ai
33 ans, j'habite en France.
```

```
Ma passion : écrire des programme de data analysis. Pour me contacter, vous pouvez envoyer un email à contact@moussakeita.fr ou contact@moussakeita.com ou bien m'appeler au 06 07 08 09 10. Vous pouvez aussi aller voir mon blog à l'adresse moussakeita-blog.fr. Bonjour et au revoir """
```

Nous allons rechercher des url dans le texte et à les transformer en liens cliquables.

Pour commencer, définissons la forme que peuvent prendre les url. Celles-ci commenceront par http ou https, seront suivies de ://, suivies des sous-domaines et du nom de domaine, ne contenant que des chiffres, lettres, tirets et points et une extension de 2 à 4 caractères, suivies enfin du chemin d'accès (facultatif) comprenant tous les caractères sauf l'espace. Voici donc une regex possible pour rechercher des url :

```
myUrlRegex=compile(r'https?://[a-zA-Z0-9\.-]+\.[a-zA-Z]{2,4}(/\S*)?')
```

Exemple 4: Convertir un ensemble de texte en dataframe en utilisant les expressions régulières (regex)

Soit le texte défini par le dictionnaire suivant.

```
textData = {'rawText': ['Arizona 1 2014-12-23       3242.0',
                'Iowa 1 2010-02-23       3453.7',
                'Oregon 0 2014-06-20       2123.0',
                'Maryland 0 2014-03-14       1123.6',
                'Florida 1 2013-01-15       2134.0',
                'Georgia 0 2012-07-14       2345.6']}
```

On souhaite convertir ce texte en un dataframe de quatre variables:state, sex, date et score.

Importons d'abord le texte brut comme dataframe

```
import pandas
df = pandas.DataFrame(textData , columns = ['rawText']) # dataframe à une seule colonne
print(df)
```

Comme on peut le constater, ce texte regorge quatre variables dont les valeurs ont été collées.

On peut distinguer les états (Arizona, Iowa,...), le sexe (1 et 0), la date (ex: 2014-12-23), les espaces et une variable numérique (score)

Il va donc falloir décomposer cette chaîne en plusieurs variables. Pour cela, on va s'appuyer sur les expressions régulières regex. La démarche sera la suivante :

Considérons qu'il y ait quatre variables: state, sex, date et score. Extrayons chacune de ces variables. Etant donné que chaque variable se présente sous un format propre à elle, on peut élaborer des expressions suivantes pour extraire les valeurs:

```
#Extraction de la variable state
df['state'] = df['raw'].str.extract('([A-Z]\w{0,})')
```

Tout caractère commençant par une lettre majuscule (de A à Z) suivi par n'imporque le caractère alphanum (o ou plusieur fois)

```
#Extraction de la variable sex(female=1)
df['female'] = df['raw'].str.extract('(\d)') # female est une variable à 1 chiffre
#Extracation de la date
df['date'] = df['raw'].str.extract('(....-..-..)')
```

La date est sous le format 4 caractères (de toute sorte)+ un tiret + 2 caractère+ 1 tiret+ 2 caractères (ici les chiffres sont traités comme caractères)

```
# Extraction de la variable score
df['score'] = df['raw'].str.extract('(\d\d\d\d\.\d)')
```

Score est une variable à 4 chiffres +.+ 1 chiffre

Chapitre 3 : Gestion de fichiers et traitements de texte sous Python

Ce stroisième chapitre vise à présenter les méthodes de gestions de fichiers et de traitement de textes sous Python. La discussion sera centrée autour de la présentation de principaux modules permettant la gestion des fichiers (ouvertures, lecture, écriture, etc..) ainsi que l'exploitation des textes.

3.1. Aperçu général sur quelques modules de gestion de fichiers sous Python

Les principaux modules permettant la gestion des fichiers sous Python sont: os et sys et shutil. Pour les charger, on fait :

```
import os, sys,shutil
```

Le module sys permet de renvoyer des informations sur le système Python utilisé. Exemples :

```
sys.path   # Liste des dossiers dans lesquels Python cherche les modules à importer. On peut alors utiliser l'indiage pour récupérer un répertoire. Exemple le répertoire courant : sys.path[0]
sys.modules  # Liste des modules importés par le programme.
sys.version # Donne des informations sur la version de Python qui est utilisée.
sys.version_info # Donne, sous forme d'un tuple, des informations sur la version de Python. Cette forme peut être plus pratique à utiliser lorsqu'il s'agit de tester si la version utilisée correspond à certains critères.
if sys.version_info[0] < 3:
    print('La version de Python que vous utilisez est trop ancienne.')
sys.float_info # Donne, sous forme d'un objet, des informations sur les nombres à virgule du système (précision, taille maximale, etc…).
sys.getfilesystemencoding() # Renvoie le nom de l'encodage utilisé par le système.
```

Le module os permet la gestion des fichiers. Par exemples ouvrir et fermer les fichiers, créer et supprimer des dossiers, créer ou modifier les répertoires de travails, etc… La section ci-dessous illustre les différentes applications du module os.

3.2. Utilisation du module o.s pour la gestion des fichiers et des répertoires

3.2.1. Définir un répertoire de travail

Au lancement de python le répertoire de travail courant est celui dans lequel se trouve l'exécutable de l'interpréteur. Sous Windows, c'est C:\Python3X, le X est la version de Python.

Pour afficher le répertoire courant, on fait

```
print(os.getcwd())
```

Pour choisir un répertoire propre à vous, on fait par exemple :

```
os.chdir("C:/PYTHON TUTORIAL/data" )
```

On remarque que ce répertoire est défini avec le slash /. Si on veut utiliser l'antislash sous Windows, il faut alors doubler le slash. Exemple

os.chdir("C:\\PYTHON TUTORIAL\\ data") # ce qui n'est pas très efficient

Pour renvoyer le répertoire courant, on fait :

```
sys.path[0]
```

3.2.2. Créer un nouveau dossier dans le répertoire défini

Par exemple créons le dossier nommé « mesdata » dans le répertoire courant.

```
os.mkdir("mesdata") # mesdata est crée dans le répertoire courant
```

la fonction mkdir() suppose, à priori, que tous les noms de dossier dans l'arborescence du path sont déjà existants. Si ce n'est pas le cas, il faut utiliser makedirs comme suit:

```
os.makedirs(path) # où path est le chemin complet
```

Cette fonction est équivalente à un appel récursif à mkdir() : elle crée récursivement tous les dossiers qui n'existe pas jusqu'au dossier final de path. Exemple :

```
os.makedirs("C:/dossier1/dossier2/dossier3")
```

3.2.3. Renommer un dossier dans le répertoire

Pour renommer un dossier, on utilise la fonction os.rename(). Exemple :

```
os.rename("mesdata","mesdata2")
```

3.2.4. Supprimer un dossier

Pour supprimer un dossier, on utilise la fonction os.rmdir(). Exemple :

```
os.rmdir("mesdata")
```

3.2.5. Tester si un fichier existe dans le répertoire

Pour tester si un fichier existe dans le répertoire courant, on peut utiliser soit la fonction exists() ou la fonction isfile() du module os ou bien la fonction is_file du module pathlib . Exemples :

Utilisation de la fonction exists() de os

```
import os.path
os.path.exists('myfile.txt')  #  renvoie  True  ou  False  selon  que  le
fichier myfile.txt existe.

#instructions conditionnelles
os.path.exists('myfile.txt'):
     print("Oui le fichier existe")
```

Utilisation de la fonction isfile() de os

```
import os.path
os.path.isfile('myfile.txt')  #  renvoie  True  ou  False  selon  que  le
fichier myfile.txt existe.

#instructions conditionnelles
if os.path.isfile('myfile.txt'):
     print("Oui le fichier existe")
```

Utilisation de la fonction is_file() de pathlib

```
import pathlib
x = pathlib.Path('myfile.txt') # définit le fichier
x.is_file()  #  renvoie  True  ou  False  selon  que  le  fichier  myfile.txt
existe.

#instructions conditionnelles
if x.is_file():
     print("Oui le fichier existe")
```

NB : La différence entre la fonction exists() et la fonction isfile() est que la fonction exists() peut renvoyer True pour un dossier lorsque celui-ci exists alors que la fonction isfile() renverra False car il ne s'agit pas d'un fichier.

3.2.6. Supprimer un fichier exitant dans le répertoire

Pour supprimer un fichier, on utilise la fonction os.remove(). Exemple :

```
os.remove("myfile.txt")
```

Généralement on élabore une suppression conditionnelle qui consiste à exécuter l'instruction de suppression lorsque le fichier existe. Exemple :

```
import os.path
if os.path.isfile('myfile.txt'):
     os.remove('myfile.txt')
```

NB: On peut aussi utilizer la function is_file() du module pathlib tel que présenté ci-dessus.

3.2.7. Lister l'ensemble des fichiers présents dans un répertoire (avec ou sans extensions)

On utilise la fonction os.listdir(). Exemple :

```
os.listdir() # Indiquer le chemin complet s'il ne s'agit pas du répertoire courant ex :os.listdir("/mydir") Attention, cette fonction listdir liste aussi les dossiers présents dans le répertoire courant.
```

Pour lister l'ensemble des fichiers ayant une extension définie (ex : .txt), on peut adopter l'une des méthodes suivantes :

```
# première méthode
import os
for file in os.listdir(): # Indiquer le chemin complet s'il ne s'agit pas du répertoire courant ex :os.listdir("/mydir")
    if file.endswith(".txt"):
        print(file)
```

```
#Deuxième méthode
import glob, os
os.chdir()
for file in glob.glob("*.txt"):
    print(file)
```

NB : Dans la première méthode, on utilise la fonction de traitement endswith() qui permet de prendre en compte tous les suffixes pour sélectionner les fichiers. On peut aussi afficher les fichiers en se basant sur des préfixes. Dans ce cas on va utiliser la fonction startswith(). Ex :

```
import os
for file in os.listdir():
    if file.startsswith("myfile"):
        print(file)
```

3.2.8. Découper le chemin complet : la fonction os.path.split()

Cette fonction décompose le path en un tuple de deux éléments le répertoire et le fichier. Exemple

```
os.path.split("D:/PYTHON TUTORIAL/data/mydata.txt")
```

3.2.9. Recomposer le chemin complet à partir d'un chemin et du nom du fichier : la fonction os.path.joint()

Cette fonction fait que le chemin inverse de split() et rassemble un répertoire et un fichier pour former un path complet. Exemple

```
os.path.join("D:/PYTHON TUTORIAL/data/, "mydata.txt")
```

3.2.10. Tester si un chemin existe

```
os.path.exists("D:/PYTHON TUTORIAL/data") # renvoie True ou False
```

3.2.11. Tester si un chemin conduit à un fichier

```
os.path.isfile("D:/PYTHON TUTORIAL/data/mydata.txt")# True ou False
```

3.2.12. Tester si le chemin indiqué est un répertoire

```
os.path.isdir("D:/PYTHON TUTORIAL/data")
```

Voir aussi

os.walk() # pour la création de fichier en parcourant le chemin de haut en bas ou de bas en haut

os.path.abspath(path) # : qui retourne le chemin absolu du chemin relatif path donné en argument. Il permet aussi de simplifier les éventuelles répétitions de séparateurs.

os.path.realpath(path) #: Retourne le chemin path en remplaçant les liens symboliques par les vrais chemins, et en éliminant les séparateurs excédentaires.

os.path.split(path) # : Sépare path en deux parties à l'endroit du dernier séparateur : la deuxième partie contient le fichier ou le dossier final, et la première contient tout le reste (c'est à dire le chemin pour y accéder). Type retourné 2-tuple of strings

os.path.basename(path) #: Retourne le dernier élément du chemin. Cela revient à appeler :func :~os.path.split et à récupérer le deuxième élément du tuple.

os.path.dirname(path) #: Retourne le chemin pour accéder au dernier élément. Cela revient à appeler :func :~os.path.split et à récupérer le premier élément du tuple.

os.path.expanduser(path) # : Si path est ~ ou ~user, retourne le chemin permettant d'accéder au dossier personnel de l'utilisateur.

os.path.join(path1[, path2[, ...]]) #: Permet de concaténer plusieurs chemins.

3.2.13. Ouvrir un fichier

D'une manière générale, pour ouvrir un fichier on utilise la fonction os.open(). Toutefois, la manière d'utiliser open() dépend selon que le fichier soit déjà existant ou pas, selon qu'il soit ouvert pour lecture seule ou pour écriture, etc...

Exemples

```
fichier = os.open("mydata.txt", os.O_RDONLY ) # Ouvrir un fichier existant pour lecture seule
fichier = os.open("mydata.txt", os.O_WRONLY ) # Ouvrir un fichier existant pour ecriture seule

fichier = os.open("mydata.txt", os.O_RDWR ) # Ouvrir un fichier existant pour lecture et écriture

fichier = os.open("mydata.txt", os.O_CREAT ) # Créer et Ouvrir un fichier si inexistant
```

NB : On peut aussi utiliser l'opérateur OU (|) pour faire l'une ou l'autre opération. Exemple: ouvrir un fichier en lecture et écriture ou créer un fichier inexistant

```
fichier = os.open("mydata.txt", os.O_RDWR|os.O_CREAT )
```

Il existe plusieurs opérations à l'ouverture d'un fichier :

os.O_RDONLY: ouvrir en lecture seule

os.O_WRONLY: ouvrir en écriture seule

os.O_RDWR: ouvrir pour la lecture et l'écriture

os.O_NONBLOCK: ouverture sans blocage

os.O_APPEND: ajoute le contenu à chaque écriture

os.O_CREAT: Créer le fichier si elle n'existe pas

os.O_EXCL: erreur de création si le fichier existe

Consulter la documentation R sur la fonction python.

Indiquer l'encodage du fichier

Très souvent, pour lire correctement un fichier, il faut spécifier l'encodage. Par exemple :

```
x = open('mydata.txt', 'r',encoding='utf-8') # lit un fichier encodé en utf-8
x = open('mydata.txt', 'r',encoding='ascii') # lit un fichier ascii
```

Nous reviendrons plus en détails sur l'encodage des fichiers plus tard.

3.3. Gestion de répertoires et de fichiers : utilisation du module shutil

Tout comme les module sys, os, le module shutil offre plusieurs fonctions de gestion de répertoires et de fichiers. Nous présentons quelques-unes de ces fonctions.

3.3.1. Copie de dossier ou de fichiers : avec le module shutil

Il existe plusieurs fonctions ayant des propriétés spécifiques pour faire les copies de fichiers. Ci-dessous les principales :

3.3.1.1. Utilisation de la fonction shutil.copyfile()

Exemples:

```
os.chdir("D: /PYTHON TUTORIAL/data" ) # fixation du répertoire
shutil.copyfile("mydata.txt","mydata2.txt") # fait une copie dans le même répertoire
shutil.copyfile("mydata.txt","D:/PYTHON TUTORIAL/mesdata2/mydata2.txt") # fait une copie dans un autre répertoire
```

3.3.1.2. Utilisation de la fonction shutil.copy

La fonction shutil.copy agit comme shutil.copyfile mais fait la copie du fichier (y compris les droits)

Exemple :

```
shutil.copy("mydata.txt","mydata3.txt") # fait une copie dans le même répertoire
shutil.copy("mydata.txt","D:/PYTHON TUTORIAL/mesdata2/mydata2.txt")  # fait une copie dans un autre répertoire
```

3.3.1.3. Utilisation de la fonction shutil.copyfileobj

La fonction shutil.copyfileobj permet de copier le contenu d'un fichier vers un autre fichier. Exemple

```
fh1 = open('mydata.txt')
fh2 = os.open('test_mydata.txt', os.O_CREAT) # créer un nouveau fichier test_mydata.txt(si inexistant)
fh3 = open('test_mydata.txt', 'w') # ouvrir le fichier créer en mode write (si déjà existant)
shutil.copyfileobj(fh1, fh3)# copier le contenu du fichier mydata au fichier test_mydata;
fh1.close(); fh3.close() # ferme les fichiers.
```

3.3.1.4. Utilisation de la fonction shutil.copystat

Cette fonction recopie à la fois les permissions et les dates de dernier accès et dernière modification présentes sur le fichier. Exemple :

shutil.copystat('myFile', 'myFile2') # recopie à la fois les permissions et les dates de dernier accès et dernière modification présentes sur le fichier myFile vers le fichier myFile2, mais sans affecter le contenu de myFile2.

A voir aussi :

shutil.copy2()

shutil.copytree

shutil.move

shutil.rmtree

3.3.2. Création de fichier archive ZIP ou TAR avec shutil

On peut créer un fichier archive avec le module shutil avec la fonction shutil.make_archive. Exemple :

```
from shutil import make_archive
#import os
shutil.make_archive('myDir', 'zip', '.', 'myDir') # création d'une archive myDir.zip
```

Le premier argument est le nom de l'archive, sans l'extension. Le 2ème argument est le type d'archive. Le 3ème argument est le directory racine de l'archive : on fait un chdir à cet endroit avant de créer l'archive (si on avait mis myDir, le .zip contiendrait le contenu sans le niveau myDir, pas terrible !). Le 4ème argument est le directory à archiver.

Il existe plusieurs autres formats d'archives possibles :

'zip' : pour les zip.

'tar' : pour les tar.

'gztar' : pour les tar.gz.

'bztar' : pour les tar.bz2.

NB : Pour désarchiver un fichier, on utilise la fonction

3.4. Visualiser le contenu du fichier lu : les fonctions read(), read(n), readline(), readlines(), et xreadlines()

3.4.1. Utilisation de la fonction read()

La fonction read() affiche la totalité du contenu du fichier en le regroupant dans une chaîne unique séparée par \n . Exemple :

Soit le fichier x défini comme suit :

```
x = open('mydata.txt', 'r',encoding='utf-8')
x.read()
```

NB : Ne pas oublier de fermer le fichier ouvert (en lecture seule) en utilisant la fonction close() comme suit :

```
x.close()
```

3.4.2. Utilisation de la fonction readline()

Soit le fichier x défini comme suit :

```
x = open('mydata.txt', 'r',encoding='utf-8')
```

La fonction readline() renvoie la première ligne du fichier x terminée par \n . Exemple :

```
x.readline()
```

Attention: avec la fonction readline(), pour renvoyer la deuxième ligne, il faut spécifier 2 fois ; pour renvoyer la troisième ligne, il faut spécifier 3 fois. Ainsi de suite. À chaque nouvel appel de readline(), la ligne suivante est renvoyée. Exemple

```
x = open('mydata.txt', 'r',encoding='utf-8')
x.readline() # renvoie la première ligne
x.readline() # renvoie ladeuxième ligne
x.readline() # renvoie la troisième ligne
...
...
x.readline() # renvoie la nième ligne
```

Cette propriété montre alors que pour renvoyer toutes les lignes, il faut indiquer autant de fois de lignes. La fonction readline() agit alors comme un itérateur (discuté dans les sections précédentes). Il peut s'avérer utile de penser à une boucle while pour afficher toutes les lignes. Exemple

```
x = open('mydata.txt', 'r',encoding='utf-8')
ch=x.readline()
```

```
while ch!="": # on peut aussi utiliser la boucle for in (voir ci-
dessous)
      print(ch)
      ch=x.readline()
x.close()
```

Remarquez aussi que lorsque l'on affiche les différentes lignes du fichier à l'aide de la boucle while et de l'instruction print, Python saute à chaque fois une ligne (mais il existe des solutions pour éviter cette situation). Nous reviendrons plus tard sur le traitement des espaces vides.

Utilisation de la boucle for pour chaque ligne d'un fichier.

```
x = open('mydata.txt', 'r',encoding='utf-8')
for i in x :
print(i)
x.close()
```

3.4.3. Utilisation de la fonction readlines()

La fonction readlines() renvoie toutes les lignes du fichier d'un sous forme de liste

```
x = open('mydata.txt', 'r',encoding='utf-8')
x.readlines()
```

3.4.4. Utilisation de la fonction read(n)

La fonction read(n) où n est un nombre entier permet de renvoyer les n premières caractères d'un texte à partir d'une position indiquée. Exemple :

```
x = open('mydata.txt', 'r',encoding='utf-8')
x.read(n) # fournit n caractères de x à partir de la position courante
```

3.4.5. Utilisation de la fonction xreadlines()

Cette fonction est généralement utilisée pour les gros fichiers. Exemple

```
f = open('HUGE.log','r'):
for line in f.xreadlines():
   print(line)
f.close()
```

3.4.6. Gestion des lignes en blank générées par les commandes readline() et readlines()

Pour éliminer les lignes vides (blanks) générées par les fonction readline() ou readlines(), on utilise les fonctions strip() qui élimine les espaces de part et d'autre d'un texte. On peu aussi utiliser la rstrip() qui élimine le texte à droit du texte. Exemples d'application :

```
x = open('mydata.txt', 'r', encoding='utf-8')
for i in x :
    print(i.strip()) # supprime l'espace de part et d'autre du texte
    print(i.rstrip()) # supprime l'espace à droite du texte
x.close()
```

3.5. Découper un texte en une liste de mots : la fonction splitlines() ou la fonction append()

Il existe plusieurs approches pour découper un texte en liste de mots. Les deux manières les plus courantes osnt l'utilisation de la fonction splitlines() ou la formation d'une liste à partir d'une boucle « for » sur les éléments de la chaine en utilisant append.

3.5.1. Utilisation de la fonction splitlines()

La fonction splitline() permet de former un objet liste dont les éléments sont constitués des lignes du fichier. Exemples :

Soit un texte mytext définit comme suit :

```
mytext = "Line1-a b c d e f\nLine2- a b c\n\nLine4- a b c d" # Généralement cette ligne provient de la lecture d'un fichier en utilisant x.read().
print(mytext.splitlines( ) ) # Renvoie: ['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
print(mytext.splitlines( 0 )) # Renvoie: ['Line1-a b c d e f', 'Line2- a b c', '', 'Line4- a b c d']
print(mytext.splitlines( 1 )) # renvoie ['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
print(mytext.splitlines( 2 )) # renvoie ['Line1-a b c d e f\n', 'Line2- a b c\n', '\n', 'Line4- a b c d']
```

On constate alors que le chiffre indiqué a un effet dans la définition du séparateur. ici, o et vide seulement permettent d'avoir le résultat escompté.

3.5.2. Utilisation de la boucle for avec la fonction append()

Exemple:

```
x = open('mydata.txt', 'r', encoding='utf-8')
mylinelist = [] # crée une liste vide (qui doit recueillir les mots)
for line in x
mylinelist .append(line.strip())
x.close()
```

Attention, lorsque le texte contient à la fois un séparateur de texte (ex; tabulation ou autre) et un séparateur de line (\n), les commandes indiquées ci-dessus ne sont plus suffisantes. Il faut alors ajouter des fonctions supplémentaires. Exemple soit le texte suivant:

```
mystring                                                                 =
"Line1\ta\tb\tc\td\te\tf\t\nLine2\ta\tb\tc\t\n\t\nLine4\ta\tb\tc\td"  #
ce texte contient à la fois \t (pour tabulation) and \n
```

Alors pour mettre chaque ligne dans un objet list, on procède en deux étapes. D'abord créer les listes avec les lignes et les tabulations

```
mystringlist=mystring.splitlines( 0 ) # pour un fichier, on utilisera
la boucle for décrite ci-dessus.
print(mystringlist)
```

Ensuite, on cherche à remplacer les tabulations \t par l'espace dans chaque élément de mystringlist. Pour cela, on va se servir du module re (regular expression) que nous étudierons plus tard.

```
import re
mystringlist = [re.split(r'\t+', i) for i in mystringlist] # pour
chaque élément i de mystringlist, on applique la fonction
re.split(r'\t+',).
print(mystringlist)
```

Cette fonction a pour rôle de décomposer chaque élément i mystringlist en des sous-éléments en faisant split. Ces sous-éléments sont présentés sous forme de list. En fin de compte, mystringlist devient une liste de listes.

Noter aussi qu'on pouvait aussi utiliser les deux fonctions patt = re.compile("[^\t]+"); patt.findall(mystring) pour formes les mêmes listes. Une fois que les séparateurs tabulation sont enlevée, il faut reconstituer les lignes en utilisant l'opérateur join() sur les sous-élements. Ainsi, on a:

```
index=len(mystringlist) # Nombre d'éléments(lignes) dans mystringlist
for k in list(range(0,index)):
    globals()["l%s"%k]=mystringlist[k] # Récupérer les sous-éléments
correspondant à k et faire join() avec espace comme séparateur
    globals()["l%s"%k]=" ".join(globals()["l%s"%k])
    mystringlist[k]=globals()["l%s"%k] # les tabulations remplacées
par des espace
print(mystringlist)
```

On peut aussi joindre les lignes entre elles pour former une seule ligne représentant tout le texte.

3.6. Déplacer le curseur dans un texte : les fonctions seek() et tell()

La fonction seek() permet de se déplacer au nième caractère d'un fichier. Et la fonction tell() permet d'afficher où en est la lecture du fichier, c'est-à-dire quel caractère (ou octet) est en train d'être lu. Exemple

```
x = open('mydata.txt', 'r')
print(x.readline())
print(x.tell())
print(x.readline())
print(x.tell())
print(x.seek(0))
print(x.readline())
print(x.readline())
print(x.readline())
print(x.seek(2))
print(x.tell())
x.close()
```

La méthode seek() permet facilement de remonter au début du fichier lorsque l'on est arrivé à la fin ou lorsqu'on en a lu une partie.

3.7. Ouvrir et modifier le contenu d'un fichier : la fonction write() et writelines()

Pour pouvoir modifier le contenu d'un fichier (ajouter ous supprimer des éléments), celui-ci doit être d'abord ouvert en mode write. Exemple :

```
x = open('mydata.txt', 'w', encoding='utf-8')
# ajoutons la ligne "lion" au fichier x
x.write('lion') # Attention pour écrie plusieurs mots, il faut indiquer plusieurs write(). Exemple:
a='girafe' ; b='tigre' ;c='singe' ; d='souris'
os.open('anim.txt', os.O_CREAT) # pour créer un nouveau fichier anim.txt (inexistant)
fic=open('anim.txt','w') # si preexistent
fic.write(a)
fic.write(b)
fic.write(c)
fic.write(d)
fic.close()
```

Attention, il est très important d'indiquer l'encoding d'un fichier lorsqu'on l'ouvre en mode write (ou même en read())

Souvent lorsqu'on veut ajouter du texte au fichier ouvert, il peut arriver des problèmes de compatibilité entre les encoding. C'est pourquoi, il faut toujours harmoniser les encoding. Pour cela, on peut utiliser utf-8 qui est compatible avec beaucoup d'autres codecs. L'exemple ci-dessus devient:

```
fic=open('anim.txt','w',encoding='utf-8')
fic.write(a)
fic.write(b)
fic.write(c)
fic.write(d)
fic.close()
```

On pouvait aussi spécifier a, b, c et d sous forme de liste et utiliser writelines(). Exemple :

```
animaux=['girafe', 'tigre', 'singe', 'souris'] # création d'une liste
fic=open('anim.txt','w',encoding='utf-8') # créer et ouvre le fichier
fic.writelines(animaux) # ajouter la liste au fichier fic.
```

Attention: avec write() et writelines() les caractères se suivent sur la même ligne sans séparateur. On peut alors ajouter un séparateur si l'on veut faire apparaitre sur la même ligne. On peut aussi utiliser l'opérateur \n si l'on veut écrire chaque texte sur une ligne spécifique.

Exemple 1 : Ecrire tous les éléments sur la même ligne avec séparateur espace

```
a='girafe' ; b='tigre' ;c='singe' ; d='souris'
fic=open('anim.txt','w',encoding='utf-8')
fic.write("%s"%a+ " ")
fic.write("%s"%b+ " ")
fic.write("%s"%c+ " ")
fic.write("%s"%d+ " ")
fic.close()
```

Exemple 2 : Ecrire de chaque élément sur une ligne spécifique avec write():

```
a='girafe' ; b='tigre' ;c='singe' ; d='souris'
fic=open('anim.txt','w',encoding='utf-8')
fic.write("%s"%a)
fic.write("\n"+ "%s"%b) # utilisation de \n
fic.write("\n"+ "%s"%c)
fic.write("\n"+ "%s"%d)
fic.close()
```

Exemple 3 : Ecrire de chaque élément sur une ligne spécifique avec writelines():

```
animaux=['girafe', 'tigre', 'singe', 'souris'] # création d'une liste
fic=open('anim.txt','w',encoding='utf-8') # créer et ouvre le fichier
animauxJoin= "\n".join(animaux)
fic.writelines(animauxJoin) # ajouter la liste au fichier fic avec
séparateur \n comme fin de ligne (actif lors du print).
fic.close()
```

Exemple 4 : Combinaison des fonctions readline() et write()

Ecrire un programme qui recopie le contenu du fichier mydata.txt dans un nouveau fichier nommé dest.txt.

On a :

```
os.open('dest.txt', os.O_CREAT) # créer un nouveau fichier dest.txt
fic1=open('mydata.txt','r',encoding='utf-8')  #  créer  et  ouvre  le
fichier mydata en lecture
fic2=open('dest.txt','w',encoding='utf-8')  #  ouvre  le  fichier  dest
écriture
ch=fic1.readline()
while ch!="":
    fic2.write(ch)
    ch=fic1.readline()
fic1.close()
fic2.close()
```

Utilisation de la boucle for

```
animaux = ['poisson', 'abeille', 'chat']
fic = open('mydata2.txt', 'w',encoding='utf-8')
for i in animaux:
    fic.write(i)
fic.close()
```

Exemple 5 : Écrire un programme qui recopie le contenu du fichier source.txt dans le fichier but.txt en supprimant toutes les lignes qui commencent par le symbole.

```
fic1=open('source.txt','r')
fic2=open('but.txt','w',encoding='utf-8')
ch=fic1.readline()
while ch!="":
    if ch[0]!="#" :
        fic2.write(ch)
    ch=fic1.readline()
fic1.close()
fic2.close()
```

Exemple 6 : Création d'un fichier HTML et stocker les résultats

Dans les exemples ci-dessus, on a utilisé les fichiers txt. On peut aussi créer des fichiers html pour recueillir les outputs. Exemple :

```
x = open('myHtmlFile.html','w') # Ouvre un fichier en mode write.
myResult = """<html>
<head></head>
<body><p>
```

```
La Data Science est une discipline de modélisation des données de
masse issues des phénomènes de l’observation…
</p></body>
</html>"""

x.write(myResult)
x.close()

# Lancement du fichier html crée
import webbrowser
webbrowser.open_new_tab('myHtmlFile.html')  #  permet  de  lancer  le
fichier dans le navigateur par défaut
```

NB : On remarque ici que l’ensemble (balises et résultats python) sont indiquées entre les triples guillemets. Cela permet à python de traiter l’ensemble comme une chaîne de caractère et le dépose comme telle dans le fichier html. C’est en ouvrant ce fichier par un navigateur que les balises seront interprétées.

Cet exemple est un cas simple. On peut le complexifier en ajoutant d’autres balises et ou faire références au valeurs d’autres variables en utilisant le formattage par l’opérateur % ou format(). Dans ce cas, il faut utiliser plusieurs instructions write() plutôt que d’utiliser les triples guillemets avec une seule write(). Exemple :

myValue='''La Data Science est une discipline technique qui associe les concepts statistiques aux algorithmes puissants de calculs informatiques en vue du traitement et de la modélisation des données de masse issues des phénomènes de l’observation (économiques, industriels, commerciaux, financières, managériaux, sociaux, etc..).'''

```
x = open('myHtmlFile.html','w') # Ouvre un fichier en mode write.
x.write("<html>") # Ecriture balise
x.write("<head></head>") # Ecriture balise
x.write("<body><p>")# Ecriture balise

x.write(myValue)# Ecriture résultats python

x.write("</p></body>")# Ecriture balise
x.write("</html>")
x.close()

# Lancement du fichier html crée
import webbrowser
webbrowser.open_new_tab('myHtmlFile.html')  #  permet  de  lancer  le
fichier dans le navigateur par défaut
```

3.8. Ouvrir un fichier en mode append 'a'

Avec le mot clé 'w', le contenu du fichier est écrasé à l'ouverture du fichier. Pour éviter cela, on doit ouvrir le fichier en mode append 'a'. Ce mode permet d'écrire à la fin du fichier sans écraser l'ancien contenu du fichier qu'on ouvre. Et lorsque le fichier n'existe pas, il est créé.

Exemple: ouvrons le fichier mydata.txt et ajoutons le mot 'lion'

```
x = open('mydata.txt', 'a')
x.write('lion') # ajoute la ligne "lion" au fichier x
x.close()
```

Attention pour écrie plusieurs mots, il faut indiquer plusieurs write(). On peut alors utiliser alors la boucle ou for. Par ailleurs, on remarque que les éléments ajoutés au fichiers sont ajoutés à la dernière ligne (mais sans espace).

3.9 Ouvrir un fichier en mode binaire 'b'

L'ouverture en mode binaire doit être combinée avec un autre mode.

```
x = open('mydata.txt', 'r' 'b')
x.readline()
x.close()
```

Ici la lecture est faite à la fois en mode read et en mode binaire

3.10. Utilisation de la fonction « with ... as » pour la gestion des fichiers

Le mot-clé with permet d'ouvrir un fichier, d'exécuter les instructions associées et ensuite de fermer un fichier sans être obligé de spécifier les fonction open() et close() . C'est manière commode d'ouverture des fichiers. Car si pour une raison ou une autre l'ouverture conduit à une erreur (problème de droits, etc), l'utilisation de with garantit la bonne fermeture du fichier. Cela signifie simplement que, si une exception se produit le fichier sera tout de même fermé à la fin du bloc (ce qui n'est pas le cas avec l'utilisation de la méthode open()). Exemple:

```
with open('mydata.txt', 'r') as x:
    for i in x:
    print(i)
```

Dans cette formulation, la ligne with open('mydata.txt', 'r') as x: est équivalente à x=open('mydata.txt', 'r')

Aussi, la boucle n'est pas terminée par x.close car à la sortie de la boucle python ferme automatiquement x. La commande with est plus générale et utilisable dans d'autres contextes (méthode compacte pour gérer les exceptions).

3.11. Encodage et décodage des textes en python

Python gère plusieurs types d'encodage de texte. Cependant utf-8 est le langage universel, il est donc conseillé de l'utiliser. Sous windows, on utilise fréquemment Latin_1 pour les caractères accentués. Pour voir la liste des encodage sous python, on fait:

```
import encodings
for code in sorted(set(encodings.aliases.aliases.values())):
    print(code)
```

Et comme certains de ces encodages ont plusieurs alias, on pourrait donc en compter plus:

```
print(len(encodings.aliases.aliases.keys()))
Pour avoir des informations sur l'encoding du système utilisé, on fait :
import sys
print(sys.getdefaultencoding()) # Pour voir le codage par défaut de python
print(sys.stdin.encoding) # l'encode des inputs (fichiers utilisés par système d'exploitation)
print(sys.stdout.encoding) # l'encodage des outputs (fichiers crées par le système)
print(sys.getfilesystemencoding()) # Encodage du système d'exploitation. Sous windows, cela renvoie 'mbcs'
```

Et Pour fixer l'encodage dans le programme, on ajoute en début de programme la ligne suivante (avec #):

```
# -*- coding : Nomencodage -*
```
où Nomencodage doit être remplacé par le type d'encoage utf8 ou Latin_1 . Exemples :

```
# -*- coding : utf-8 -*
# -*- coding : Latin-1 -*
```

En Python 3, le codage utf-8 est automatique. Toutes les chaînes sont par défaut de type 'str'

Il faut savoir que pour toute opération de manipulation de fichier (reading, writing), deux opérations sont souvent nécessaires: encoding et decoding. On utilise l'encoding lorsque l'on a obtenu un fichier de résultats et qu'on veut l'enregistrer sur le disque. On utilise le décoding, lorsqu'on a un fichier input et qu'on souhaite faire d'autres opérations. Nous allons donc détailler ci-dessous ces deux opérations.

3.11.1. Décodage de texte

Pour décoder un texte reçu, on utilise la fonction decode(). La syntaxe du décoage est la suivante:

```
monTexte.decode('nomencoding','errorstype')
```

Où monTexte est le texte à décoder ; nomEncoding est le nom de l'encodage dans lequel se trouve déjà le texte. Le paramètre errorstype prend par défaut la valeur 'strict' qui signifie que l'encodage doit soulever unqiement des erreurs liés aux type Unicode. Les autres valeurs sont 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace'.

Exemple1 : Obtenir une chaîne 'unicode' depuis une chaîne 'str' encodée en utf8 :

Soit le texte suivant:

```
une_chaine = 'Chaîne' # cette valeur est déja encodée en UTF8, donc la chaine est en UTF8
type(une_chaine) # donne <type 'str'>
une_chaine = une_chaine.decode('utf8') # met le texte en unicode.
```

Attention l'attribut decode n'est disponible que quand le texte en question est déjà encodé.

```
type(une_chaine) # donne <type 'unicode'>
```

Exemple 2

```
mystr = "this is string example....wow!!!";
mystr1  = mystr1.encode('ascii','strict'); # on va d'abord l'encoder en byte avant de le décoder.
```

On a:

```
mystr2=mystr1.decode('ascii','strict')
print ("Encoded String: " + str(mystr1) ) # renvoie: b'this is string example....wow!!!'
print ("Decoded String: " + mystr2) # this is string example....wow!!!
```

Noter aussi qu'on peut utiliser le module codecs pour effectuer le décodage d'un fichier. Ou plutôt ouvrir un fichier dans le codec indiqué. Exemple:

```
import codecs# le module codecs est installé par défaut avec python.
f = codecs.open("file.txt", "r", "utf-8") # ouvre le fichier en mode read "r" initialisé encodé en utf-8.
```

Il existe diverses autres fonctions associées à codecs. Consulter par exemple la page https://docs.python.org/2/library/codecs.html

3.11.2. Encodage de texte

L'encodage est la démarche inverse du décodage qui consiste à partir d'un texte avec un codage donné pour le mettre sous un autre codage. Sa syntaxe est la même que pour le décodage en remplaçant simplement decode par encode. Exemple:

mystring = "this is string example....wow!!!"; # un texte de codage par défaut str.

mystring=mystring.encode('cp037','strict')

print (mystring) # donne: b'\xa3\x88\x89\xa2@\x89\xa2@\xa2\xa3\x99\x89\x95\x87@\x85\xa7\x81\x94\x97\x93\x85KKKK\xa6\x96\xa6ZZZ'

Ainsi pour décoder ce texte et le mettre sous codage str, on fait:

mystring=mystring.decode('cp037','strict')

print (mystring)

NB: un texte ne peut avoir avoir à la fois l'attribut encode et decode à la fois. C'est soit l'un ou l'autre.

3.11.3. Peut-on détecter l'encodage d'un fichier texte ?

Dans l'optique d'une manipulation de texte, la seule difficulté pour décoder un texte est de savoir quelle est le codage d'origine. Malheureusement, il n'y aucune règle qui permet de déterminer avec précision l'encodage d'un texte (à moins que celui qui a été crée le fichier indique). Par exemple, pour les pages html ou xml, le type d'encoding est souvent fournit en en-tête (ex: charset="utf-8"). C'est ce que l'on doit utiliser pour décoder le texte avant de l'utiliser le coding d'un texte. Il faut donc faire extrêment attention lorsqu'on veut décoder un fichier. Il faut s'assurer que le codec en question est bien le bon.

Pour détecter l'encodage d'un fichier texte, il n'y a pas de régles pour déterminer l'encodage d'un fichier. D'abord, une manière plus simple serait de copier un caractère dans le texte et de rechercher dans les bases de normes comme utf-8, ISO, etc... Sinon on peut faire quelques remarques comme suit (à utiliser avec précautions):

Normalement, il y a au début du fichier un BOM (byte order mark), c'est à dire une séquence d'octets:

- UTF8 : 3 premiers octets = EF BB BF
- UTF-16/UCS-2 (big-endian) : 2 premiers octets = FE FF
- UTF-16/UCS-2 (little-endian) : 2 premiers octets = FF FE
- ASCII/AINSI : pas de BOM, directement le contenu.

En UTF-8, les caractères sont généralement codés sur 1, 2 ou 3 octets (cela peut être plus, mais c'est rare). Ce codage respecte les règles suivantes :

codage sur 1 octet : 0xxxxxxx

codage sur 2 octets : 110xxxxx 10xxxxxx

codage sur 3 octets : 1110xxxx 10xxxxxx 10xxxxxx (représentation en binaire des octets. "x" signifiant valeur quelconque)

Connaissant tous ces éléments, il suffit de parcourir le fichier pour voir si les règles sont respectées. Par exemple si un octet commence par 110????? Alors l'octet suivant commence par 10??????. A la première violation des règles, on peut arrêter de parcourir le fichier et supposer que ce n'est pas de l'UTF-8.

Pour l'ASCII c'est facile : toutes les valeurs sont inferieure à 128, c'est à dire que tous les octets sont de la forme 0xxxxxxx. D'ailleurs les formats UTF-8, ANSI et ISO sont compatibles avec l'ASCII. (ANSI : organisme de standardisation americain ; ISO : organisme de standardisation international).

3.12. Présentation du module URLLIB pour gestion des urls et la lecture de fichiers à partir des urls (uniform resource locator)

En Python 3, l'ancien module urllib a été divisé entre trois modules nommés urllib.request, urllib.parse et urllib.error. Nous allons présenter chacun de ces modules ci-après.

3.12.1. Le module urllib.request

D'une manière générale, le module URLIB.REQUEST() permet d'envoyer de requêtes. On distingue deux types de requêtes: les requêtes GET et les requêtes POST. Les requêtes GET permettent de récupérer des informations à partir d'une page et les requêtes POST envoient des infos vers les pages. Pour les requêtes de type GET, plusieurs fonctions sont associées au module URLLIB.REQUEST(): urlopen(), urlretrieve() ou Request(). Chacune de ces fonctions sont détaillées ci-dessous (Consulter aussi le lien https://docs.python.org/3.0/library/urllib.request.html pour plus de détail).

3.12.1.1. Utilisation de la fonction urllib.request.urlopen()

La fonction urllib.request.open() permet de capturer une page web comme un fichier (à l'image de la fonction open pour les fichiers d'autres types). Exemples: Téléchargeons le texte se trouvant à l'url : http://www.pdb.org/pdb/files/1BTA.pdb

```
import urllib.request
u = urllib.request.urlopen("http://www.pdb.org/pdb/files/1BTA.pdb") # On pouvait aussi utiliser la commande with ... as u (comme pour un fichier classique)
mytexte = u.read() # # la fonction read() permet de lire tout le contenu du fichier u. On pouvait aussi utiliser readline() ou readlines()
u.close()
mytexte=mytexte.decode('ascii','strict') # On va décoder le texte du ascii(byte) vers str(unicode)  (le décodage ici permet de rendre lisible le texte et de l'harmoniser)
```

Il est également possible d'enregistrer le texte téléchargé dans un autre fichier d'un autre type (txt, csv, etcc). Par exemple : écrire du texte mytexte dans un fichier nommé mypdbfile.pdb

```
f = open("mypdbfile.pdb","w", encoding='utf-8')
f.write(mytexte)
f.close()
Afficher le contenu de mypdbfile.pdb
x = open('mypdbfile.pdb', 'r', encoding='utf-8')
ch=x.readline()
while ch!="": # on peut aussi utiliser la boucle for ... in
    print((ch).strip())
    ch=x.readline()
x.close()
```

L'exemple ci-dessus lit un texte brut. On peut aussi lire une page web balisée (html ou xml). Voir exemples ci-dessous:

```
import urllib.request
response=urllib.request.urlopen('http://python.org/')
texte=response.read()
print(texte) # nous verrons plus tard comment exploiter un texte avec balisage html ou xml
```

En cas de présence d'espace \n, on peut ajouter .strip() comme pour les fichiers classiques.

```
texte.close()
```

Remarquons qu'avec le module urllib, il ne s'agit pas simplement de lire une page html avec urlopen() mais aussi tout texte se trouvant à un url qui peut être du texte ou autre données. Nous reviendrons sur le traitement des pages html plus bas.

3.12.1.2. Utilisation de la fonction urllib.request.urlretrieve()

La fonction urlretrieve() permet de récupérer le fichier url et le stocke dans un fichier local temporaire pour une utilisation future. Exemple:

```
import urllib.request
mylocaFile, headers = urllib.request.urlretrieve('http://python.org/')
# enregistrement dans mylocalFile
texte = open(mylocaFile) # lecture de mylocaleFile pour utilisation: read(), readline(s), etc...
```

3.12.1.3. Utilisation de la fonction urllib.request.Request()

La fonction Request() permet d'envoyer la requête de l'utilisateur sous forme de requête HTTP au serveur qui lui repond en ayant reçu le message. Voici-ci dessous un exemple d'utilisation:

```
import urllib.request
```

```
req = urllib.request.Request('http://www.voidspace.org.uk') # Envoie de la requête HTTP enregistré sous le nom req.
with urllib.request.urlopen(req) as response: # Lecture de la page renvoyée par la requête en utilisant urlopen()
   the_page = response.read()
# Noter aussi qu'on peut utiliser Request() avec d'autre protocole autre que http comme par exemple FTP. Exemple :
req = urllib.request.Request('ftp://example.com/')
```

3.12.1.4. Cas des requêtes POST

Pour ce qui concerne les requêtes POST, elles se présentent comme des requêtes GET, à la seule différence qu'on doit indiquer les infos à envoyer. Exemple:

```
import urllib.parse # Importation du module parse() que nous allons étudier plus bas
import urllib.request
url = 'http://www.someserver.com/cgi-bin/register.cgi' # serveur auquel, on va envoyer les requetes post
values = {'name' : 'Michael Foord', 'location' : 'Northampton','language' : 'Python' } # définition des données à envoyer
data = urllib.parse.urlencode(values) # encodage des valeurs
data = data.encode('ascii') # Mise des données en encodage "byte"
req = urllib.request.Request(url, data)  # Envoi de la requêtes
with urllib.request.urlopen(req) as response: # lecture
   the_page = response.read()
```

3.12.2. Utilisation du module requests comme alternatif de urllib.request

Toutes les étapes indiquées ci-dessous notamment les requêtes GET ou les requêtes POST peuvent être fait avec le module requests directement. Voici ci-dessous un exemples:

```
import requests
# requête get
mytext = requests.get('https://www.python.org') # récupeération de la page
if 'Python is a programming language' in mytext.content: # Vérifie si le texte indiqué est présent dans mytext.
    print("Cette phrase exite sur la page"
else :
    print("Cette phrase n'existe pas sur la page")
# requête post
mydico = dict(key1='value1', key2='value2') # définit les données à poster
mypost = requests.post('http://httpbin.org/post', data=mydico) # envoie une requête post
```

```
print(mypost.text) # affiche les données envoyées: ici c'est le dictionnaire.
```

3.12.3. Le module urllib.error

Le module urllib.error permet de gérer les exceptions renvoyée par le module urlib.request(). Il existe deux fonctions associées à urllib.error que sont notamment URLError() et HTTPError. Chacune de ces fonctions sont décrites ci-dessous (Consulter aussi la page https://docs.python.org/3.1/library/urllib.error.html pour plus de détails).

3.12.3.1. Utilisation de la fonction urllib.error.URLError()

Une requête peut envoyer des erreurs liées à l'URL par exemple problème d'accès, problème de connexion, serveur inexistant ,typeError, ValueError, etc.. Il faut alors utiliser des exceptions pour donner plus de précisions sur le type d'erreur. On utilise alors la fonction URLError qui renvoie des précisions sur l'erreur rencontrée. L'exemple ci-dessous permet de renvoyer un tuple contenant le code de l'erreur et le message d'erreur:

```
req = urllib.request.Request('http://www.pretend_server.org') # envoi de la requête
try: urllib.request.urlopen(req) # ouverture de la page
except urllib.error.URLError as e: # Récupérer et affichier l'error en cas d'erreur
    print(e.reason)     # Cela renvoie:   [Errno 11004] getaddrinfo failed
```

3.12.3.2. Utilisation de la fonction HTTPError()

Cette fonction permet de gerer des exceptions liées au protocole http. Voir les détails https://docs.python.org/3.1/library/urllib.error.html

3.12.4. Le module urllib.parse()

Le module urllib.parse() est l'outil pour faire du parsing d'un url pour récupérer ou poster différentes informations. L'utilisation la plus classique du module urllib.pars()e est de décomposer un url en des composantes ou bien rassembler différentes composantes pour former un url (consulter la page: https://docs.python.org/3/library/urllib.parse.html)

3.12.4.1. Décomposer un url en différentes composantes

La syntaxe pour réaliser cette opération est la suivante:

```
urllib.parse.urlparse(urlstring, scheme='', allow_fragments=True)
```

urlstring représente le lien internet au complet ; scheme='' est le nom qu'on donne à chaque composante de ce lien. Par exemple: //netloc/path;parameters?query#fragment signifie que le lien sera décomposé en fonction netloc, path, parameter, query et fragment.

```
Exemple:
import urllib.parse
mylink =
urllib.parse.urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
print(mylink) # donne ParseResult(scheme='http',
netloc='www.cwi.nl:80', path='/%7Eguido/Python.html', params='',
query='', fragment='')
On obtient alors un objet ParseResult. Ce qui pourait ensuite être
utilisé à d'autres fins telle que la conversion en list ou tuple.
Exemple:
print(list(mylink)) # donnne :['http', 'www.cwi.nl:80',
'/%7Eguido/Python.html', '', '', '']
print(tuple(mylink)) # donnne :('http', 'www.cwi.nl:80',
'/%7Eguido/Python.html', '', '', '')
```

Attention: urlparse reconnait netloc seulement si elle est proprement introduite par le symbôle "//". Sinon, l'input est considéré comme un url relatif. Dans ce cas le parsing commencera au niveau de la composante path. Exemple

```
import urllib.parse
print(urllib.parse.urlparse('//www.cwi.nl:80/%7Eguido/Python.html')) #
renvoie: ParseResult(scheme='', netloc='www.cwi.nl:80',
path='/%7Eguido/Python.html', params='', query='', fragment='')
print(urllib.parse.urlparse('www.cwi.nl/%7Eguido/Python.html')) #
Renvoie: ParseResult(scheme='', netloc='',
path='www.cwi.nl/%7Eguido/Python.html',params='', query='',
fragment='')
print(urllib.parse.urlparse('help/Python.html')) # Renvoie:
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
query='', fragment='')
```

Noter aussi qu'une autre variante de urlparse est urlsplit().

3.12.4.2. Rassembler différentes composantes en un url

Pour rassembler différentes composantes en un url, on utilise la fonction urlunparse() ou bien la fonction lunsplit() ou encore la fonction geturl()

Il existe plusieurs autres fonctions associée à urllib.parse dont notamment: parse_qs(),parse_qsl(), urljoin(), urldefrag() mais surtout plus important urlencode(). En effet urlencode() permet d'encoder les informations en byte afin d'éviter des erreurs d'harminsations. Consulter la documentation python sur urlencode().

3.13. Extraction et gestion des textes a partir des pages html ou xml (pages avec balisages) : utilisation du module BeautifulSoup

Le module urllib étudié précédemment a montré comment on peut obtenir du texte à partir d'un url notamment avec urlopen(). A présent, nous allons étudier d'autres outils permettant d'exploiter les textes à partir des pages html, xml, etc… Nous allons nous servir du module urllib notamment le module requests (pour l'ouverture des url) et du module BeautifulSoup pour faire du parsing de la page et ensuite utiliser des fonctions de gestion de texte pour extraire les textes. Le module Beautifulsoup qui doit être d'abord téléchargé (voir les instructions d'installation des modules dans la section guide d'installation package).

3.13.1. Utilisation du module BeautifulSoup: extraction du texte à partir d'une page htlm ou xlm avec parsing

3.13.1.1. Lecture de la page html ou xml

BeautifulSoup est un module permettant de naviguer dans les balises d'une page html ou xml pour renvoyer les informations souhaitées en utilisant un parser.

Exemple introductif: Nous allons télécharger la page suivante http://en.wikipedia.org/wiki/List_of_A_Song_of_Ice_and_Fire_characters en utilisant les outils classiques de lecture d'url comme urllib.request.urlopen() ou requests.

```
import requests
url                                                                  =
'http://en.wikipedia.org/wiki/List_of_A_Song_of_Ice_and_Fire_character
s'
r = requests.get(url) # Récupérer la page html.
myHtmlText  =  r.text  #  Utilisation  de  la  fonction  text  qui  récupère
tout le contenu de la page
r.close()
```

On pouvait aussi utiliser urllib pour lire le contenu

```
import urllib
r                                                                    =
urllib.request.urlopen("http://en.wikipedia.org/wiki/List_of_A_Song_of
_Ice_and_Fire_characters")
myHtmlText =r.read()
r.close()
```

3.13.1.2. Le parsing de la page récupérée

Une fois qu'on a récupéré la page, l'étape suivante est de faire du parsing pour récupérer les différentes informations définies par les balises à l'intérieur de la page. Le parsing a pour but de démeler les balises des autres élments de la page. Le but final étant de récupérer le contenu réel de la page. On se sert alors des outils de BeautifulSoup comme suit :

```
from bs4 import BeautifulSoup
myParsedText = BeautifulSoup(myHtmlText , 'html.parser') # faire du parsing sur le code de la page récupérée.
```

'html.parser' est le parser utilisé ici. Il est installé par défaut. Il existe plusieurs autres parser. Par exemple: on peut aussi utiliser lxml comme parser (après avoir installé)

Ci-dessous les parsers couramment utilisés pour le parsing d'une page web avec python

Parser	Utilisation	observation
html.parser	BeautifulSoup(markup, "html.parser")	installé par défaut sous python
xml	BeautifulSoup(markup, "lxml")	module lxml à installer
lxml-xml	BeautifulSoup(markup, "lxml-xml")	module lxml à installer
lxml-xml	BeautifulSoup(markup, "lxml-xml")	module lxml à installer

Exemples :

```
myParsedText = BeautifulSoup(myHtmlText, 'lxml') # transforme le code du format html au format lxml de  BeautifulSoup
print(myParsedText)
print(myParsedText.prettify()) # Afficher le contenu avec indentation: écrit de manière lisible le code en ecrivant les balises ouvrantes et fermantes sur des lignes séparées du contenu de la balise
print(myParsedText.get_text()) ###   Extraire tout le texte brut se trouvant sur la page (sans les balises).
```

Toutefois get_text laisse des grands espaces dans le texte. C'est pourquoi, il faut faire des traitements supplémentaires pour avoir un texte propre. Pour ces traitements, regarder l'exemple de code qui extrait le texte de la page html sans les balises, les scripts et les espaces (plus bas).

Par ailleurs, avant d'appeler get_text, il peut s'avérer nécessaire de supprimer les scripts et les styles présents sur la page parsée. Pour cela, on fait:

```
for i in myParsedText(["script", "style","form","noscript"]): # on peut allonger cette liste
    i.extract()    # enlever les scripts, les styles, etc..
print(myParsedText.get_text())
```

Souvent les éléments à supprimer sont (ces élements proviennent du guide du module patter-web, il faut voir comment les adapter à un traitement direct):

- Strip javascript: supprime tous les élements <script>
- Strip CSS: supprime tous les élements <style>
- Strip comments: supprime tous les élements <!-- -->
- Strip forms: supprime tous les élements <form>
- Strip tags: supprime tous les HTML tags.
- Collapse spaces: remplace les espaces consecutives par un simple espace.
- Collapse linebreaks: remplace les coupures de lignes consecutives par une coupure unique
- Collapse tabs: remplace les espaces tabulations consecutives par un simple espace.

Il existe également d'autres variantes d'utilisation de get_text(). Exemple:

```
text = myParsedText.get_text(separator=' ') # qui prend le blank comme séparator
```

On peut aussi utiliser

```
text = myParsedText.body.get_text() # si l'on veut exclure les textes qui viennent de la base head()
```

Il faut lire sur la compréhension de la structure des pages html ou xml notamment les balisages (voir plus bas les principales balises)

3.13.1.3. Navigation dans le texte parsé

Une fois que le parsing est fait et que l'objet soup est crée alors, on peut naviguer maintenant dans le texte. Voici ci-dessous quelques exemples :

Exemple 1 :

```
print(myParsedText.title) # Affichage du titre du document
print(myParsedText.title.name) # Affichage nom du titre
print(myParsedText.title.string) # Affichage du texte du titre sans le mot clé title
print(myParsedText.title.parent.name) # base parent de title. Ici c'est head
print(myParsedText.p) # Affichage du premier paragraphe
```

```
print(myParsedText.a) # Affichage du premier lien hypertexte
print(myParsedText.find_all('a')) #afficher tous les liens hypertexte
ou url (a) présents sur la page
print(myParsedText.find(id="link3")) # Recherche la ligne pour
laquelle id=link3
```

Trouver tous les liens (toutes les parentheses de type 'a' qui ont une cle 'href') et créer une liste

```
liens = []
for mytag in myParsedText.find_all('a'):
    if mytag.has_attr("href"):
    liens.append(mytag["href"])
print(liens)
```

Trouver le premier tag où la clé id prends valeur PageTop

```
myParsedText.find(id="PageTop")
```

Déplacer le curseur au bon endroit: recherche par type de tag et paires (clé,valeur) du dictionnaire associé.

```
noeud = myParsedText.find_all('body')[0]
print(noeud)
```

Se déplacer à la première parenthèse à l'interieur de la parenthèse actuelle.

```
premier_fils = list(noeud.children)[0]
```

Se deplacer à la premiere parenthèse contenant la parenthèse actuelle.

```
parent = noeud.parent
```

Extraction de tous les urls se trouvant sur la page

```
for i in myParsedText.find_all('a'): # Trouver tous les tags de type
'a'
    print(i.get('href'))
```

Exemple 2 : Code qui extrait le texte de la page html sans les balises, les scripts et les espaces

```
import urllib.request
from bs4 import BeautifulSoup
url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"
myHtmlText = urllib.request.urlopen(url).read()
myParsedText = BeautifulSoup(myHtmlText, 'html.parser')
for i in myParsedText(["script", "style"]): # kill all script and
style elements
    i.extract()    # rip it out (enlever les scripts et les styles)
cleanText = myParsedText.get_text() # get text
cleanText=cleanText.splitlines() # break into lines (La fonction
splitline permet de former une liste formées des lignes du fichier)
```

```
lines = (line.strip() for line in cleanText) # remove leading and
trailing space on each
phraseList = (phrase.strip() for line in lines for phrase in
line.split("  ")) # break multi-headlines into a line each
cleanText = '\n'.join(phrase for phrase in phraseList  if phrase) #
drop blank lines and join phrases with \n as separator
print(cleanText)
```

3.13.2. Quelques balises html (tags)

Ci-dessous la description de quelques balises couramment rencontrées sur une page html ou xml.

<a> lien hypertexte

<b> ou <strong> texte en gras

<blockquote> retrait de texte

<code> code informatique

<frameset> frames ou cadres

<font> balise de texte

<hr> trait horizontal

<td> cellule de tableau

<i> ou <em> texte en italique

<img> insertion d'une image

 élement de liste

 liste

<p> paragraphe

<strike>ligne au travers d'un texte

<style> feuille de style

<sub> chiffre en indice

<sup> chiffre en exposant

<table> tableau

<td> cellule de tableau

<tr> ligne de tableau

<tt> texte télétype

<u> ligne en dessous d'un texte

 liste à puces

Remarque : cas des meta-balises

Les meta-tags, meta-balises ou metas-données sont des balises situées dans l'en-tête du code html, entre les balises <head> et </head>. Ces balises sont composés d'informations descriptives sur le site, elles seront analysées par les robots pour permettre une indexation dans les moteurs de recherche c'est pourquoi elles doivent être correctement remplies. Les méta-balises de référencement sont:

- title :titre du site ou de la page

<title>Le titre du site</title>

-description: description du site en une phrase de 200 caractères

<meta name="description" content="La description du site">

-keywords : mots clés du site

<meta name="keywords" content="les, mots, clés">

-identifier-url: adresse du site

<meta name="identifier-url" content="http://www.lesitecom">

-author : auteur du site

<meta name="author" content="auteur">

-robots: diriger les robots(suivra ici la page d"index puis suivantes)

<meta name="robots" content="index, follow">

-revisit-after: revisite du robot en jours

<meta name="revisit-after" content="10 days">

-copyright: copyright du site

<meta name="copyright" content="lauteur»">

NB : La balise meta keywords peut contenir 1000 caractères, la balise meta description ne doit pas contenir plus de 200 caractères

3.13.3. Utilisation du module pattern-web pour le parsing des pages web

Le module pattern-web est un module très puissant de traitement de texte des pages html et xml. Il permet de faire avec beaucoup d'efficacité l'extraction des textes tout en éliminant les balises, les scripts, les styles, etc... Il propose aussi plusieurs fonctions utiles: telle que les recherches sur les moteurs de recherche (google, etc..). On peut trouver plus de détails sur la page: http://www.clips.ua.ac.be/pages/pattern-web .

3.13.4. Utilisation du module xml pour le parsing des pages web

Noter aussi qu'on peut utiliser le module lxml pour récuprer directement le contenu d'une page sans passer par BeautifulSoup. L'exemple suivant est une illustration.

import lxml

import requests

mypage = requests.get('http://econpy.pythonanywhere.com/ex/001.html')

contenu = lxml.html.fromstring(mypage.content) # Il faut utiliser mypage.content plutôt que mypage.text car html.fromstring s'attend implicitement à des caractères de type byte

print(contenu)

3.13.5. Utilisation du module html2text pour le parsing des pages web

Le module html2text permet également de lire des pages html. Exemple :

import html2text # ce module doit être installé d'abord

text = html2text(myHtmlText)

3.13.6. Utilisation du module NLTK pour le parsing des pages web

Le module NLTK sur lequel nous reviendrons plus tard permet aussi de lire des pages html. Exemple :

```
import nltk  # ce module doit être installé d'abord
from urllib.request import urlopen
url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"
myHtmlText = urlopen(url).read()
text = nltk.clean_html(myHtmlText)

print(text)
```

3.14. Text mining avec le module NLTK

Dans cette section, nous allons revoir quelques procédures de traitement de textes en se basant sur le module NLTK (Natural Language Processing Toolkit).

Avant de commencer, installons d'abord un ensemble de textes présents dans la fonction book. Pour insatller les fonctions de nltk, on ouvre la fenêtre de téléchargement comme suit:

```
nltk.download()
```

Ainsi on peut télécharger et installer ainsi les modules souhaitables y compris book.

Ci-après, nous allons présenter quelques applications pratiques des fonctions du module NLTK.

3.14.1. Convertir un texte en format nltk

Très souvent pour pouvoir utiliser les fonctions de NLTK, les textes doivent être convertis d'abord dans un format adpaté reconnaissable comme tel par NLTK. Cela permet de les rendre compatibles avec certaines fonctions de nltk. Pour convertir un texte, on utilise la fonction Text comme suit:

Soit le texte raw suivant:

```
raw  = """The little pig saw the wolf climb up on the roof and lit a roaring fire in the fireplace and\
          placed on it a large kettle of water.When the wolf finally found the hole in the chimney he crawled down\
          and KERSPLASH right into that kettle of water and that was the end of his troubles with the big bad wolf.\
          The next day the little pig invited his mother over . She said &amp;amp;quot;You see it is just as I told you. \
          The way to get along in the world is to do things as well as you can.&amp;amp;quot; Fortunately for that little pig,\
          he learned that lesson. And he just lived happily ever after!"""
```

Pour mettre ce texte sous format Text nltk, on fait:

```
import nltk
from nltk.text import Text
import nltk.text
myText = nltk.text.Text(raw)
```

myText peut ainsi être utilisé pour d'autres opérations. Toutefois, cette conversion n'est pas nécessaire dans tous les cas.

3.14.2. Découper une chaîne de caractères en des mots : la fonction tokenize()

La fonction tokenize() permet de décompser un texte en une liste de mots.

Exemple 1: Découpage d'une chaine de caractères

```
montexte  = """The little pig saw the wolf climb up on the roof and lit a roaring fire in the fireplace and\
          placed on it a large kettle of water.When the wolf finally found the hole in the chimney he crawled down\
          and KERSPLASH right into that kettle of water and that was the end of his troubles with the big bad wolf.\
          The next day the little pig invited his mother over. She said &amp;amp;quot;You see it is just as I told you. \
          The way to get along in the world is to do things as well as you can.&amp;amp;quot; Fortunately for that little pig,\
          he learned that lesson. And he just lived happily ever after!"""
```

On découpe ce texte en une liste comme suit

```
from nltk.tokenize import word_tokenize
liste_texte = word_tokenize(montexte)
print(liste_texte)
```

Toutefois, le module NLTK considère les ponctuations comme des caractères à part entière. Pour les exclure, on utilise la fonction escape du module re (voir un exemple d'application suivant).

Exemple 2 : Découpage d'une liste de chaine de caractères en une liste de mots excluant les ponctuations

Soit le texte suivant qui est une liste composé de trois élements.

```
montexte = ["We are attacking on their left flank but are losing many men.",
               "We cannot see the enemy army. Nothing else to report.",
               "We are ready to attack but are waiting for your orders."]
```

On va décomposer cette liste en une liste des listes c'est à dire un array de mots.

Pour cela, on va utiliser le module tokenizer nltk (qui devrait être installé d'abord). Il faut aussi installer le module Punkt sentence tokenization

```
from nltk.tokenize import word_tokenize
liste_texte = [word_tokenize(texte) for texte in montexte ]
```

```
print(liste_texte)
```

On constate que la liste des mots considère les ponctuations notamment le point (.) ou les virgules comme un mot à part entière. Il faut corriger cette situation en excluant les ponctuations
Ainsi, on va traiter cette situation en utilisant les expressions régulières. On suit les étapes suivantes:

```
import re
import string
regex = re.compile('[%s]'%re.escape(string.punctuation)) # Définition d'une expression régulière qui exclut les ponctutaions.
liste_texte_sans_punctuation = []
for review in liste_texte:
    new_review = []
    for token in review:
        new_token = regex.sub(u'', token)
        if not new_token == u'':
            new_review.append(new_token)
    liste_texte_sans_punctuation.append(new_review)
print(liste_texte_sans_punctuation)
```

NB: la fonction tokenize() peut aussi s'appliquer à un texte brut qui n'était pas déjà sous forme de liste comme dans le cas précédent.

Consulter les pages http://docs.python.org/2/library/string.html (https://docs.python.org/2/library/re.html#re.escape)

3.14.3. Rechercher un mot dans un texte et afficher la (les) phrase(s) qui le contient(nent)

Pour chercher un mot dans un texte avec le module NLTK, on utilise la fonction concordance(). Par exemple, pour rechercher le mot monstrous dans le texte text1 et renvoyer les lignes correspondantes, on fait:

```
from nltk.book import*  # importation des textes préconçus de nltk
text1.concordance("monstrous")
```

NB: En examinant les occurences renvoyées, on peut se rendre compte du contexte général dans lequel ce mot est souvent employé. Par exemple, on remarque que le mot size et le mot picture sont les plus fréquents. Cela veut donc dire que le mot monstrous survient souvent dans un contexte de size ou de picture. Cela permet donc d'avoir une idée des premières associations entre les mots dans un texte.

Attention dans l'exemple ci-dessus, nous avons utilisé un exemple de texte préconçu avec NLTK. Pour utiliser notre propre texte, nous devons d'abord utiliser la fonction Text de NLTK. Voici ci-dessous 3 exemples d'application

Exemple 1: Concordance avec le mot 'wolf'

```
import nltk
from nltk.text import Text
import nltk.text
mytexte1  = """The little pig saw the wolf climb up on the roof and lit a roaring fire in the fireplace and\
          placed on it a large kettle of water.When the wolf finally found the hole in the chimney he crawled down\
          and KERSPLASH right into that kettle of water and that was the end of his troubles with the big bad wolf.\
          The next day the little pig invited his mother over . She said &amp;amp;quot;You see it is just as I told you. \
          The way to get along in the world is to do things as well as you can.&amp;amp;quot; Fortunately for that little pig,\
          he learned that lesson. And he just lived happily ever after!"""

## Créer une liste de tokens
myTokens = nltk.word_tokenize(mytexte1)
## Créer un texte de tokens
myTokenText = nltk.text.Text(myTokens)
### Afficher les phrases de concordance
print(myTokenText.concordance('wolf'))
```

Exemple 2: Concordance avec le mot 'système'

```
mytexte2 = "Un IDE ou \"environnement de développement\" est un logiciel \
constitué d'outils qui facilitent l'écriture et les tests dans un \
langage défini, voire plusieurs.\
\nCet IDE comporte en général un éditeur avec coloration syntaxique,\
un système de gestion de fichiers (sauvegarde/chargement),\
un compilateur, un exécuteur de programme, un système d'aide en ligne,\
des indicateurs de syntaxe etc. \
\n Le plus connu est peut être Éclipse."

## Créer une liste de tokens
myTokens = nltk.word_tokenize(mytexte2)
## Créer un texte de tokens
myTokenText = nltk.text.Text(myTokens)
### Afficher les phrases de concordance
print(myTokenText.concordance('système'))
```

Exemple 3: Concordance avec un mot provenant d'un fichier texte.

Pour ce cas, il faut d'abord importer le texte en utilisant la fonction read(). Ensuite suivre la même méthode que dans les deux exemples précédents.

Ex: soit le fichier myConcordanceTextFile.txt. Renvoyons les phrases qui contiennent le mot "Bonjour". On a:

```
with  open('myConcordanceTextFile.txt',  'r',  encoding='utf-8')  as  x:
#(fichier déjà dans le répertoire de travail)
    mytexte3=x.read() # récupération du texte.
## Créer une liste de tokens
myTokens = nltk.word_tokenize(mytexte3)
## Créer un texte de tokens
myTokenText = nltk.text.Text(myTokens)
### Afficher les phrases de concordance
print(myTokenText.concordance('Bonjour'))
```

NB : avec la fonction concordance(), il est seulement possible d'afficher les phrases mais ces phrases ne sont pas directement stockabe dans un objet. Il faut alors penser à élaborer une fonction basée sur la fonction ConcordanceIndex qui récupère les indices des caractères et ensuite extraire les caractères correspondants pour former les phrases.

Exemple :

1-Construction de la fonction

```
def    phrasesFunct(mot_cible,    phrase_cible,    left_margin    =    10,
right_margin = 10):
   ## Création des tokens
    tokens = nltk.word_tokenize(phrase_cible)

    ## Création du texte des tokens
    text = nltk.text.Text(tokens)
     ## Récupération de tous les indices du mot cible
    c = nltk.ConcordanceIndex(text.tokens, key = lambda s: s.lower())
  ## Récupération de tous les intervalles d'indices du mot cible en
utilisant text.tokens[start;end]. La fonction map est utilisée pour
que quand  la position offset – l'intervalle cible < 0, la valeur par
défaut sera fixée à zero
    concordance_txt   =   ([text.tokens[map(lambda   x:   x-5   if   (x-
left_margin)>0  else  0,[offset])[0]:offset+right_margin]for  offset  in
c.offsets(mot_cible)])
  ## joindre les phrases pour chacun de la phrase cible
    return  [''.join([x+'  '  for  x  in  con_sub])  for  con_sub  in
concordance_txt]
```

2-Test de la function

Soit le texte suivant :

```
raw  = """The little pig saw the wolf climb up on the roof and lit a
roaring fire in the fireplace and\
```

```
          placed on it a large kettle of water.When the wolf finally found the hole in the chimney he crawled down\
          and KERSPLASH right into that kettle of water and that was the end of his troubles with the big bad wolf.\
          The next day the little pig invited his mother over. She said &amp;amp;quot;You see it is just as I told you. \
          The way to get along in the world is to do things as well as you can.&amp;amp;quot; Fortunately for that little pig,\
          he learned that lesson. And he just lived happily ever after!"""
 tokens = nltk.word_tokenize(raw)
myText = nltk.text.Text(tokens)
print(myText)
myText.concordance('wolf') # default text.concordance output
print("")
print ('Résultat de la fonction')
results = phrasesFunct('wolf', raw)
for result in results:
    print (result)
```

3.14.4. Repertorier les n premiers mots les plus fréquents dans un texte avec la fonction FreqDist

Pour répertorier par exemple les n premiers mots les plus fréquents, on utilise la fonction FreqDist de nltk. Ex:

```
fdist1 = FreqDist(text1) # donne la liste de tous les mots avec leur fréquence respective
words=list(fdist1.keys()) # la liste de tous les mots
words=words[:50] # les 50 premiers mots
print(words)
freqwords=list(fdist1.values()) # listes des fréquences
freqwords=freqwords[:50] # les 50 premières fréquences
print(freqwords)
```

On peut aussi représenter la fonction de densité et de répartition empirique des 50 mots les plus fréquents. Alors on fait:

```
fdist1.plot(50, cumulative=False) # Densité
fdist1.plot(50, cumulative=True) # répartition
```

NB: ne pas oublier d'utiliser les fonctions lower() et upper() afin de regrouper les mots écrit en majuscule ou en minsucule.

```
text1_maj=[w.upper() for w in text1]
```

3.14.5. Compter le nombre d'occurences de la longueur des mots dans un texte

Au lieu de compter le nombre de mots dans un texte, on peut aussi compter le nombre d'occurence du nombre de caractères dans un texte. L'idée est la suivante.

```
long=[len(w) for w in text1] # compter le nombre de  caractères de chaque mot présent dans le texte text1
fdist = FreqDist(long) # calculer la fréquence de chaque nombre calculé
list(fdist.keys()) # renvoyer les nombres calculés
list(fdist.values()) # renvoyer les fréquence respectives
print(fdist.items()) # liste des items et leur fréquence
```

NB: ne pas oublier d'utiliser les fonctions lower() et upper() afin de regrouper les mots écrits en majuscule ou en minsucule. Ex:

```
fdist = FreqDist(word.lower() for word in word_tokenize(sent)) # exemple d'utilisation
print((fdist.freq('monstrous'))) # Le pourcentage de fréquence du mot monstrous dans le total de text1
print(fdist.max()) # la fréquence la plus élevée
```

3.14.6. Identifier des collocations de mots

D'une manière générale, une collocation de mots est un pair de mots qui s'accompagnent souvent. Par exemple: long term , non smoker, United States, United Nations. Pour voir l'ensemble des collocations présentes dans text4 on fait:

print(text4.collocations())

Il faut noter qu'on peut nous même définir nos propres collocations de mots. Pour cela, on utilise la fonction bigrams. Exemple:

```
from nltk import bigrams
mycolloc=bigrams(['more', 'is', 'said', 'than', 'done'])
print(list(mycolloc))
```

3.14.7. Stemming, lemmatisation et post-tag des mots dans un texte

3.14.7.1. Le stemming ou la racinisation

Le stemming (ou la racinisation) est la réduction d'un mot à racine, c'est-à-dire sa forme la plus réduite. Il s’agit d’enlever soit le préfixe, soit le suffixe pour le réduire à son strict minimum. Cette racine n'est pas nécessairement un mot bien défini, mais elle peut être utilisée pour générer des mots en concaténant un suffixe à droite ou un préfixe à gauche. Par exemple, les mots anglais fish, fishes and fishing ont tous la même racine fish, qui est un mot correct. De l'autre côté, les mots study, studies et studying se racinisent studi, qui n'est, en fait, pas un mot anglais.

Le plus souvent, les algorithmes de stemming (les stemmers) sont basés sur des règles de désuffixation. L'exemple le plus célèbre est le stemmer de Porter, introduit dans les années 1980 et actuellement mis en œuvre dans une variété de langages de programmation.

Traditionnellement, les moteurs de recherche et d'autres applications IR (Information Retrieving) appliquent des stemming pour améliorer la chance de correspondre aux différentes formes d'un mot, les traiter comme des synonymes, comme ils "appartiennent" à un même ensemble.

Avec le module NLTK, il existe plusieurs fonctions de stemming. Les plus utilisées PorterStemmer(), SnowballStemmer(), le stemmer de Lancaster LancasterStemmer(). Pour voir la liste complète des stemmers de NLTK, consulter la page http://www.nltk.org/api/nltk.stem.html. Ci-dessous quelques exemples d'application :

```
# SnowballStemmer()

from nltk.stem.snowball import SnowballStemmer

stemmer2 = SnowballStemmer("english") # à adapter dans le cas du français

print( stemmer2.stem("studying"))
```

```
# PorterStemmer()

from nltk.stem import PorterStemmer

stemmer1 = PorterStemmer()

print(stemmer1.stem("studying"))
```

```
# LancasterStemmer()

from nltk.stem.lancaster import LancasterStemmer

st = LancasterStemmer()

>st.stem('maximum')      # 'maxim'

st.stem('presumably')  # 'presum'

st.stem('multiply')    # 'multiply'

st.stem('provision')   # 'provid'

st.stem('owed')        # 'ow'

st.stem('ear')         # 'ear'

st.stem('saying')      # 'say'
```

```
st.stem('crying')      #  'cry'

st.stem('string')      # 'string'

st.stem('meant')       # 'meant'

st.stem('cement')      # 'cem'
```

NB : Les stemmers de Porter et de Lancaster sont tous deux spécifiques à l'anglais.

3.14.7.2. La lemmatisation

Le but de la lemmatisation est de regrouper différentes variantes d'un mot en un terme plus englobant appelé lemme. Le processus de lemmatisation est quelque peu semblable au stemming, car il mappe plusieurs mots en une racine commune. Toutefois, le résultats de la lemmatisation est toujours un mot propre, Contrairement à celui du stemming qui n'est pas forcement un mot bien précis. Par exemple, un lemmatiser appliqué sur les mots anglais gone, going et went renvoie renvoie chacun le mot go qui est le lemme de tous ces mots.

Il faut toutefois noter que pour un meilleurs résultats, la lemmatisation nécessite de connaître le contexte d'un mot. En effet, le processus repose sur le principe que le mot est soit un nom, un verbe, etc.

Pour effectuer la lemmatisation avec le module NLTK, on peut utiliser la fonction WordNetLemmatizer(). Exemple: on veut lemmatiser le mot studying. Les étapes sont les suivantes :

```
from nltk.stem import WordNetLemmatizer

lemmatiser = WordNetLemmatizer() # définition du lemmatiseur

print(lemmatiser.lemmatize("studying"))

print(lemmatiser.lemmatize("studying", pos="v")) # l'option pos="v" signifie que le mot à lemmatiser est un verbe.

print(lemmatiser.lemmatize("students", pos="n")) # lemmatisation d'un nom
```

Pour plus de détails sur les lemmatisers de NLTK, consulter la page http://www.nltk.org/api/nltk.stem.html.

3.14.7.3. Les étiquettes grammaticales (les post-tags)

A l'inverse du stemming et de la lemmatisation, le post-tag d'un mot vise à déterminer son étiquette grammatical (nom, verbe, pronom, déterminants, adjectifs, prépositions, conjonctions et adverbes).

La fonction permettant de faire le post tag sous NLTK est pos_tag(). Ci-dessous un exemple d'utilisation.

Soit le texte myText défini comme suit :

```
myText = "This is a simple sentence"
```

On va appliquer le post tag sur chacun des élements de cette phrase. Pour cela, on va découper la phrase en des tokens et ensuite appliquer la fonction post_tag(). On a :

```
from nltk.tokenize import word_tokenize

myTokens = word_tokenize(myText) # convertit le text en tokens

from nltk import pos_tag

myTokens_pos = pos_tag(myTokens)

print(myTokens_pos)# renvoie [('This', 'DT'), ('is', 'VBZ'), ('a', 'DT'), ('simple', 'JJ'), ('sentence', 'NN')]
```

3.14.7.4. Traitement des StopWords (les mots vides)

Les stop words (mots vides) sont des mots très communement utilisés dans un langage à telle fréquence qu'il est inutile de les accorder une importance particulière lors de l'exploitation des informations contenues dans un texte, de l'indexation ou la recherche. En français, des mots vides évidents sont entre autres « le », « la », « de », « du », « ce », etc...

Dans le module NLTK, la fonction nltk.corpus.stopwords.words() permet de renvoyer l'ensemble des stopwords présents dans un corpus sous forme de liste. Par exemple pour l'anglais et le français, on a :

```
import nltk
stpw_english = nltk.corpus.stopwords.words('english')
stpw_french = nltk.corpus.stopwords.words('french')
```

Selon les besoins de l'analyse, il peut parfois être judicieux de supprimer l'ensemble des stop word dans un texte avant d'en extraire les informations essentielles à travers les mots clés ou avant de faire du texte matching ou du texte clustering.

3.15. Text Matching : calcul des Edit distances et des ratios de matching entre deux séquences de caractères

Il existe actuellement plusieurs modules pour le text matching sous python. Les plus courants sont difflib,Levenshtein, Distance ainsi que le module fuzzywuzzy. Dans cette section, nous allons présenter chacun de ces méthodes à travers quelques exemples d'application.

3 .15.1. Le module difflib

Le module difflib permet la comparaison de deux séquences de chaines de caractères. Il est conçu à partir de l'algorithme de Ratcliff/Obershelp. Les principales fonctions sont :SequenceMatcher(), differ(), ndiff(), unified_diff, context_diff(), HtmlDiff()

3 .15.1.1. Utilisation de la fonction SequenceMatcher()

La fonction SequenceMatcher() permet de calculer le taux de matching entre deux séquences de valeurs qui peuvent être de n'importe quels types (numériques et/ou alphabétiques). En indiquant la fonction ratio, on peut obtient la valeur du ratio de matching entre les deux séquences.

Exemple 1 : Comparaison entre deux chaînes et calcul du ratio de matching

```
import difflib
mymatch = difflib.SequenceMatcher(None, "NEW YORK METS", "NEW YORK
MEATS")
mymatch.ratio() # Renvoie le ratio de matching. Ici 0.962
```

Exemple2 :Soient les deux textes text1 et text2 défini comme suit :

```
text1 = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Integer
eu lacus accumsan arcu fermentum euismod. Donec pulvinar porttitor
tellus. Aliquam venenatis. Donec facilisis pharetra tortor.  In nec
mauris eget magna consequat convallis. Nam sed sem vitae odio
pellentesque interdum. Sed consequat viverra nisl. Suspendisse arcu
metus, blandit quis, rhoncus ac, pharetra eget, velit. Mauris
urna. Morbi nonummy molestie orci. Praesent nisi elit, fringilla ac,
suscipit non, tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate tristique
enim. Donec quis lectus a justo imperdiet tempus."""

text2 = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Integer
eu lacus accumsan arcu fermentum euismod. Donec pulvinar, porttitor
tellus. Aliquam venenatis. Donec facilisis pharetra tortor. In nec
mauris eget magna consequat convallis. Nam cras vitae mi vitae odio
pellentesque interdum. Sed consequat viverra nisl. Suspendisse arcu
metus, blandit quis, rhoncus ac, pharetra eget, velit. Mauris
urna. Morbi nonummy molestie orci. Praesent nisi elit, fringilla ac,
suscipit non, tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Duis vulputate tristique enim. Donec quis lectus a justo
imperdiet tempus. Suspendisse eu lectus. In nunc. """
```

Calculons le taux matching entre ces deux textes. On a :

```
import difflib
mymatch = difflib.SequenceMatcher(None, text1, text2)
mymatch.ratio() # renvoie 0.869
```

3 .15.1.2. Utilisation des fonctions get_matching_blocks() et get_opcodes() après SequenceMatcher()

Après l'exécution de la fonction SequenceMatcher, on peut utiliser des fonctions supplémentaires comme get_matching_blocks() et get_opcodes() .
La fonction get_matching_blocks() permet de renvoyer les positions où il y a matching entre les deux séquences. Quant à la fonction get_opcodes() permet de savoir comment changer la première séquence pour obtenir la seconde séquence. Exemples :

```
import difflib
# Execution du SequenceMatcher
s = difflib.SequenceMatcher(None, "private Thread currentThread;",
"private volatile Thread currentThread;")
# Exécution de get_matching_blocks
print(s.get_matching_blocks()) # renvoie [(0, 0, 6), (6, 15, 23), (29,
38, 0)] (exécuter 2 fois)
for block in s.get_matching_blocks():
     print("Text1[%d] and Text2[%d] match for %d elements" % block)
# Exécution de get_matching_blocks
print(s.get_opcodes()) # [('equal', 0, 6, 0, 6), ('insert', 6, 6, 6,
15), ('equal', 6, 29, 15, 38)]
for opcode in s.get_opcodes():
    print("%6s Text1[%d:%d] Text2[%d:%d]" % opcode)
```

NB : Il faut noter que le dernier tuple renvoyé par .get_matching_blocks () est toujours un dummy, (len (Text1), len (Text2), 0), et c'est le seul cas où le dernier élément de tuple (nombre d'éléments appariés) est 0.

3 .15.1.3. Utilisation de la fonction differ()

La fonction differ() de difflib permet de faire ressortir la dissemblance entre deux séquences.
La sortie produite se décrit comme suit :
Les lignes précédées de - indiquent celles qui se trouve dans la première séquence, mais pas la seconde.
Les lignes précédées de + sont celles qui sont dans la deuxième séquence, mais pas la première.
Si une ligne a une différence incrémentale (caractère de plus ou de moins) entre les versions, une ligne supplémentaire préfixée par « ? » est utilisé pour mettre en évidence le changement dans la nouvelle version.
Si une ligne n'a pas changé, elle est imprimée avec un espace vide supplémentaire sur la colonne de gauche afin qu'il soit aligné avec les autres lignes qui peuvent avoir des différences.

Pour comparer deux textes, il est recommandé de le découper en une séquence de lignes individuelles et passer ces séquences à la fonction comparer ().

Exemple: Soient les deux textes text1 et text2 définis précédemment.
Découpons ces textes en lignes comme suit:

```
text1_lines = text1.splitlines()
text2_lines = text2.splitlines()
```

Et appliquons la fonction compare() sur ces deux listes de lignes. On a :

```
import difflib
diff = difflib.Differ().compare(text1_lines, text2_lines)
print ('\n'.join(diff))
```

3 .15.1.4. Utilisation de la fonction ndiff()

La fonction ndiff () produit essentiellement la même sortie que differ.compare (). Mais, la fonction est spécifiquement adaptée pour le traitement des données en texte et d'éliminer le «bruit» dans les inputs.
Exemple: Soient les deux listes de lignes de textes définies précédemment text1_lines et text2_lines. La fonction ndiff() s'applique comme suit :

```
import difflib
diff = difflib.ndiff(text1_lines, text2_lines)
print ('\n'.join(list(diff)))
```

3 .15.1.5. Utilisation de la fonction unified_diff()

Alors que la classe Differ() affiche toutes les lignes d'entrée, la fonction unified_diff n'inclut que des lignes modifiées avec un peu de contexte. Exemple :

```
import difflib
diff = difflib.unified_diff(text1_lines, text2_lines, lineterm='')
print ('\n'.join(list(diff)))
```

3 .15.1.6. Utilisation de la fonction context_diff()

Cette fonction produit le même output proche de celui de unified_diff(). Exemple :

```
import difflib
diff = difflib.context_diff(text1_lines, text2_lines, lineterm='')
print ('\n'.join(list(diff)))
```

3 .15.1.7. Utilisation de la fonction Htmldiff()

La fonction HtmlDiff() produit une sortie en format HTML avec les mêmes informations que celles renvoyée par Diff(). Exemple :

```
import difflib
d = difflib.HtmlDiff()
print (d.make_table(text1_lines, text2_lines)) # produit la table de
sortie
```

3 .15.2. Le module Levenshtein

Le module Levenshtein permet de calculer l'edit distance entre deux séquences tout en renvoyant le ratio de matching. Ce module est basé sur l'algorithme de matching Levenshtein. Levenshtein a quelques points communs avec la fonction SequenceMatcher de Difflib. Toutefois, il ne prend en charge que les chaînes de caractères, pas les types de séquence arbitraire contrairement à SequenceMatcher. Cependant il reste beaucoup plus rapide. Ci-dessous quelques exemples d'application.

Exemple 1 :

```
import Levenshtein
Levenshtein.ratio('hello world', 'hello') # renvoie 0.625
```

Exemple 2 : Soient les deux textes:

```
text1 = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Integer
eu lacus accumsan arcu fermentum euismod. Donec pulvinar porttitor
tellus. Aliquam venenatis. Donec facilisis pharetra tortor.  In nec
mauris eget magna consequat convallis. Nam sed sem vitae odio
pellentesque interdum. Sed consequat viverra nisl. Suspendisse arcu
metus, blandit quis, rhoncus ac, pharetra eget, velit. Mauris
urna. Morbi nonummy molestie orci. Praesent nisi elit, fringilla ac,
suscipit non, tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate tristique
enim. Donec quis lectus a justo imperdiet tempus."""

text2 = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Integer
eu lacus accumsan arcu fermentum euismod. Donec pulvinar, porttitor
tellus. Aliquam venenatis. Donec facilisis pharetra tortor. In nec
mauris eget magna consequat convallis. Nam cras vitae mi vitae odio
pellentesque interdum. Sed consequat viverra nisl. Suspendisse arcu
metus, blandit quis, rhoncus ac, pharetra eget, velit. Mauris
urna. Morbi nonummy molestie orci. Praesent nisi elit, fringilla ac,
suscipit non, tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Duis vulputate tristique enim. Donec quis lectus a justo
imperdiet tempus. Suspendisse eu lectus. In nunc. """
```

Calcul du ratio sur les textes brut

```
Levenshtein.ratio(text1, text2) # renvoie 0.9365928189457601
```

Calcul des ratio sur le texte découpé en lignes et converties en textes

```
Levenshtein.ratio(str(text1_lines),   str(text2_lines))   #   renvoie
0.9380014587892049
```

NB: Levenshtein ne s'applique que sur les string. Alors quand on dispose d'une liste, il vaut mieux faire join d'abord et récupérer le texte.

3.15.3. Le module distance

Ce module fournit des fonctions qui permettent de calculer les edit distance entre deux séquences et renvoie le ratio de matching. Les principales fonctions disponibles dans le module distances sont Levenshtein() qui calcule la distance de Levenshtein, Hamming() qui calcule la distance de Hamming, Jaccard() et Sorensen() qui calculent respectivement la distance de Jaccard et de Sorensen. Ci-dessous les exemples d'application de ces fonctions.

3.15.3.1. Distance de Levenshtein

```
import distance
distance.levenshtein("this is a test", "this is a test!") # renvoie 1 renvoie la distance entre les deux argument (distance en nombre absolu)
```

NB la distance levenshtein peut être normalisée en utilisant l'option normalized=True. On obtient alors les ratios. Cela permet la comparabilité avec les autres mesures de distance. Ex :

```
distance.levenshtein("this is a test", "this is a test!", normalized=True) # renvoie 0.06666666666666667
```

La distance normalisée de levenshtein peut aussi se calculer avec la fonction nlevenshtein(). Cependant, deux stratégies sont disponibles pour le calcul: soit la longueur de l'alignement le plus court entre les séquences est prise comme facteur ou bien la longueur du plus long est prise comme facteur. Ex :

```
distance.nlevenshtein("this is a test", "this is a test!", method=1)
# alignement plus court comme facteur. renvoie 0.06666666666666667
distance.nlevenshtein("this is a test", "this is a test!", method=2)
# alignement plus long comme facteur. Renvoie 0.06666666666666667
```

3.15.3.2. Distance de Hamming

```
import distance
distance.hamming("this is a test?", "this is a test!") # exige que les textes soient de meme longueur. renvoie 1 (nombre absolu); distance normalisable
distance.hamming("this is a test?", "this is a test!", normalized=True)
```

3.15.3.3.Distance de Sorensen

```
import distance
```

```
distance.sorensen("this is a test", "this is a test!") # renvoie directement la distance en ratio. ici 0.06666666666666665 . Pour avoir le ratio de matching, on fait 1-distance
```

3.15.3.4.Distance de jaccard

```
import distance
distance.jaccard("this is a test", "this is a test!")# renvoie directement  la distance en ratio. ici 0.125 . Pour avoir le ratio de matching, on fait 1-distance
```

NB: Contrairement au module Levenshtein, le module distance peut s'appliquer sur une liste de string.

3.15.4. Le module FuzzyWuzzy

Le module FuzzyWuzzy est un module développé à partir des modules de bases. Il se propose d'apporter quelques améliorations à ceux-ci en intégrant quelques options supplémentaires. Une présentation détaillée de ce module se trouve sur le lien suivant : http://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/
Les principales fonctions proposées par FuzzyWuzzy sont :ratio(),token_sort_ratio(),token_set_ratio ou process(). Le choix de la méthode de matching dependra de la formulation du problème par l'utilisateur.

Exemple d'application

```
from fuzzywuzzy import fuzz, process
# Simple ratio
fuzz.ratio("this is a test", "this is a test!") # Simple Ratio     97
# partial ratio
fuzz.partial_ratio("this is a test", "this is a test!") # Partial Ratio     100
# token_sort_ratio
fuzz.ratio("fuzzy wuzzy was a bear", "wuzzy fuzzy was a bear") # Token Sort Ratio     91
fuzz.token_sort_ratio("fuzzy wuzzy was a bear", "wuzzy fuzzy was a bear") #     100
fuzz.token_sort_ratio("fuzzy was a bear", "fuzzy fuzzy was a bear") # Token Set Ratio     84
# token_set_ratio
fuzz.token_set_ratio("fuzzy was a bear", "fuzzy fuzzy was a bear") # 100
# using process
choices = ["Atlanta Falcons", "New York Jets", "New York Giants", "Dallas Cowboys"] # Process
process.extract("new york jets", choices, limit=2) #   [('New York Jets', 100), ('New York Giants', 78)]
process.extractOne("cowboys", choices) #   ("Dallas Cowboys", 90)
```

3.15.5. Comparaison des différentes méthodes de matching

Soient les deux textes suivantes:

```
text1 = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Integer
eu lacus accumsan arcu fermentum euismod. Donec pulvinar porttitor
tellus. Aliquam venenatis. Donec facilisis pharetra tortor.  In nec
mauris eget magna consequat convallis. Nam sed sem vitae odio
pellentesque interdum. Sed consequat viverra nisl. Suspendisse arcu
metus, blandit quis, rhoncus ac, pharetra eget, velit. Mauris
urna. Morbi nonummy molestie orci. Praesent nisi elit, fringilla ac,
suscipit non, tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Suspendisse eu lectus. In nunc. Duis vulputate tristique
enim. Donec quis lectus a justo imperdiet tempus."""

text2 = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Integer
eu lacus accumsan arcu fermentum euismod. Donec pulvinar, porttitor
tellus. Aliquam venenatis. Donec facilisis pharetra tortor. In nec
mauris eget magna consequat convallis. Nam cras vitae mi vitae odio
pellentesque interdum. Sed consequat viverra nisl. Suspendisse arcu
metus, blandit quis, rhoncus ac, pharetra eget, velit. Mauris
urna. Morbi nonummy molestie orci. Praesent nisi elit, fringilla ac,
suscipit non, tristique vel, mauris. Curabitur vel lorem id nisl porta
adipiscing. Duis vulputate tristique enim. Donec quis lectus a justo
imperdiet tempus. Suspendisse eu lectus. In nunc. """
```

Calculons les ratios avec différentes méthodes

```
import difflib,Levenshtein,distance
from fuzzywuzzy import fuzz

diffl    = difflib.SequenceMatcher(None, text1, text2).ratio() # ratio
Ratcliff-Obershelp
lev1     = Levenshtein.ratio(text1, text2) # ratio Levenshtein calculé
par le module Levenshtein
lev2       = 1-distance.levenshtein(text1, text2, normalized=True) #
ratio Levenshtein calculé par le module distance
#ham      = 1-distance.hamming(text1, text2, normalized=True) # ratio
de matching de hamming (utilisable uniquement quand les deux séquences
ont même longueur).
sorens     = 1 - distance.sorensen(text1, text2) # ratio de matching
de sorensen
jac      = 1 - distance.jaccard(text1, text2) # ratio de matching de
jaccard
# Affichage
fuz=fuzz.partial_ratio(text1, text1) # partial ratio fuzzywuzzy
print("Ratcliff-Obershelp ratio: ",round(diffl,3) )
```

```
print("Levenshtein ratio 1: ",round(lev1,3))
print("Levenshtein ratio 2: ",round(lev2,3))
print("Sorensen ratio: ",round(sorens,3))
print("Jaccard ratio: ",round(jac,3) )
print("fuzzywuzzy partial ratio: ",round(fuz,3))
```

NB :Les fonction présentées ci-dessous permettent de tester uniquement le matching. Elles ne permettent pas de tester "semantic similarity". Ce test nécessite l'utilisation d'autres modules.

3.16. Text Clustering : regroupement des termes d'un texte par des algorithmes de clustering

Dans cette section, nous présentons quelques algorithmes de clustering des textes en partant des exemples pratiques. Deux cas sont étudié : l'utilisation de l'algorithme « Affinity Propagation » et l'utilisation des méthodes traditionnelles de clustering comme le « k-mean clustering », « Multidimensional scaling », « Hierarchical clustering », etc. Il faut simplement noter que chacune de ces méthodes de clustering sont mises en œuvre à partir des matrices de distance (entre les mots ou en entre les textes encore appelé document). La particularité des méthodes de text clustering réside alors dans le calcul des matrices de distance.

3.16.1. Text clustering avec l'algorithme de l'Affinity Propagation

Cette méthode de clustering vise à regrouper les mots d'un texte en formant des clusters selon le principe de l'affinité entre les mots. C'est un algorithme itératif qui repose sur le partage des « affinités » :

Chaque élément c repère dans son voisinage un élément qui lui ressemble suffisamment, et augmente son affinité pour cet élément ; Les étapes suivantes consistent à « propager » cette affinité :

Chaque élément c repère celui pour qui il a la plus grande affinité, noté m ; Il ajoute à ses propres affinités celles de m ; Cette étape est répétée un certain nombre de fois, ou bien jusqu'à ce que le nombre d'éléments passe en dessous d'un certain seuil - ou encore quand cette étape n'apporte plus aucun changement.

Il y a alors trois cas :

L'élément considéré possède une affinité maximale pour un autre élément : il lui ressemble ;

L'élément considéré possède une affinité maximale pour lui-même : il est « exemplaire » (exemplar) ;

L'élément considéré possède une affinité nulle : il est « isolé ».

Le nombre d'éléments exemplaires dépend de nombreux paramètres et ne peut être donné a priori.

On obtient à l'issue de l'algorithme un arbre complet, reliant les éléments semblables qui ont pu être identifiés comme tels.

Exemple : Soit le texte suivant :

```
myText = """Polytechnique Montréal est l'une des trois plus grandes facultés d'ingénierie au Canada et la plus grande au Québec. Depuis sa fondation en 1873, cette institution d'enseignement de langue française forme des ingénieurs qualifiés. Ses diplômés ont pris part à la plupart des grands travaux de génie du Québec au xxe siècle comme la construction des grands barrages."""
```

L'algorithme de propagation d'affinité se met en œuvre comme suit :

```
import numpy as np
import sklearn.cluster
import distance
# préparation du texte
myTextWords =myText.split(" ") #découper le texte en liste
myTextWords= np.asarray(myTextWords) #Transformer la liste en array
#Calcul de la matrice de distance (ou matrice de similarité)
lev_similarity = -1*np.array([[distance.levenshtein(w1,w2) for w1 in myTextWords] for w2 in myTextWords]) # Calcul des negative euclidean distances i.e similarity matrix (multiplié par -1)
#Mise en œuvre du clustering à partir de la matrice
affprop = sklearn.cluster.AffinityPropagation(affinity="precomputed", damping=0.5) # spécification de l'algorithme et ses paramètres
affprop.fit(lev_similarity) # Extimation du modèle
clusterid=affprop.fit_predict(lev_similarity, y=None) # Générer les valeurs des clusters (méthode 1)
clusterid=affprop.labels_ # deuxième  méthode pour générer les valeurs des clusters (méthode 2)
# predict(X) permet de faire de la prévision sur un nouveau echantillon(nouvelle matrice de similarité) à partir des paramètres estimés
clusterCenterIndex=affprop.cluster_centers_indices_  # Renvoie les indices des centre des clusters (dans myTextWords)
# Affichage des textes de clusters (exemplars) et les mots qui y appartiennent
print("******Cluster word: Values******")
print()
```

```
for cluster_id in   list(range(len(clusterCenterIndex))):
    exemplar = myTextWords[clusterCenterIndex[cluster_id]] # Récupère
le centroide
    cluster_elements                                              =
np.unique(myTextWords[np.nonzero(clusterid==cluster_id)])         #
Sélectionne tous les indices où clusterlab==cluster_id et renvoie les
éléments correspondants
    print(exemplar)
    print(cluster_elements )
    cluster_elements_join = ", ".join(cluster_elements) # fait du join
    print(" -%s: %s" % (exemplar, cluster_elements_join))

# Création d'un dataframe pour récupérer les clusters et leur éléments
import pandas
clusterData    =    pandas.DataFrame(index=range(0),    columns    =
['cluster_id',  'cluster_labels',    'cluster_elements'])  #  création
dataframe vide
for cluster_id in   list(range(len(clusterCenterIndex))): # pareil
    exemplar = myTextWords[clusterCenterIndex[cluster_id]] # Récupère
le centroide
    cluster_elements=
np.unique(myTextWords[np.nonzero(clusterid==cluster_id)])         #
Sélectionner tous les indices où clusterlab==cluster_id et renvoie les
éléments correspondants
    cluster_elements_join = ", ".join(cluster_elements) # fait du join
    dico    =    {'cluster_id':    [cluster_id],    'cluster_labels':
[exemplar],'cluster_elements': [cluster_elements_join]} # dictionnaire
du cluster
    cluster_idData = pandas.DataFrame(dico, columns = ['cluster_id',
'cluster_labels',  'cluster_elements']) # dataframe du cluster
    clusterData  =  pandas.concat([clusterData,  cluster_idData])  #
appending
pandas.set_option('expand_frame_repr', False)
print(clusterData)
```

La particularité de la méthode de l'affinity propagation est qu'on a pas besoin de fixer le nombre de clusters à priori. Il permets de dégager les mots clés (centroides) d'un texte autours desquels gravitent les autres mots. Ce qui constitue un bon départ pour mieux comprendre la structure d'un texte. Toutefois, l'une de ses limites c'est qu'il ne permet pas de distinguer les StopWords qui pour la plupart du temps n'apportent pas assez d'informations dans l'analyse du texte. Les algorithmes que nous verrons par la suite permettent ce genre de traitement car ils sont basés sur la matrice TF-IDF (Term Frequency-Inverse Document Frequency).

3.16.2. Text clustering par des algorithmes basés sur la matrice TF-IDF

Les algorithmes de text clustering comme le « k-mean clustering », « Multidimensional scaling » ou le « Hierarchical clustering » sont généralement mis en œuvre à partir des matrices de similarités basées sur la matrice TF-IDF.

Le TF-IDF (Frequency-Inverse Document Frequency) est une méthode de pondération des mots permettant d'évaluer l'importance d'un terme contenu dans un document, relativement à une collection de document ou un corpus. Le poids d'un mot augmente proportionnellement au nombre d'occurrences du mot dans le document mais inversement en fonction de la fréquence du mot dans l'ensemble des document ou du corpus.

3.16.2.1. Construction de la matrice TF-IDF

Avant de mettre en œuvre le clustering, il faut d'abord calculer la matrice TF-IDF associée à la problématique étudiée. Pour mieux comprendre le concept de matrice TF-IDF, partons de l'exemple développé par Brandon Rose (http://brandonrose.org/clustering) concernant l'analyse des 100 films les plus populaires de l'histoire du Cinéma. Chaque film est décrit par un titre (title) et un synoptique, présentation permettant de saisir d'un seul coup d'œil un ensemble d'informations liées au film (synopses). Voir http://www.imdb.com/list/ls055592025/. Ces deux informations ont été déjà extraites et stockées dans des fichiers nommés respectivement title_list.txt et synopses_list_wiki.txt enregistrés dans le répertoire courant. En somme, on dispose de 100 titres auquels correspondent 100 synoptiques.

Le but visé ici est d'exploiter les synopses à travers du text clustering afin de tirer l'essentiel des informations qui sous-tendent l'ensemble de ces films et au final regrouper les films selon les grandes catégories. En gros, il s'agit de faire du text clustering sur l'ensemble des 100 films afin de dégager des grandes catégories de films plus homogènes.

Dans la perspective de la construction de la matrice TF-IDF, il faut avoir à l'esprit que chaque synopsis constitue un document; chaque document document étant constitué d'un ensemble de termes (mots).

Avant d'aborder le calcul de la matrice TF-IDF, importons d'abord les textes et effectuons les traitements nécessaires.

Importation des textes

```
########### Lecture de la liste des synopses
with open("synopses_list_wiki.txt", 'r', encoding="utf-8") as x: #
    synopses=x.read().strip() # lecture du fichier de synopses (lire tout le fichier en un seul bloc)
```

```
synopses =synopses.split("BREAKS HERE") #découper le texte dont le séparateur est le mot "BREAKS HERE"
print(len(synopses))
for i in range(len(synopses)):
    synopses[i]=synopses[i].strip() # Suppression des lignes vides représentées par \n
    #print(synopses[i])
del synopses[len(synopses)-1] # Suppression du dernier élément qui contient uniquement l'espace vide " "
################### Lecture de la liste des titres
with open("title_list.txt", 'r', encoding="utf-8") as x: #
    titles=x.read().strip() # lecture du fichier des titres(lire tout le fichier en un seul bloc)
titles =titles.split("\n") #découper le texte dont le séparateur est le mot "\n" nouvelle ligne
#print(len(titles))
for i in range(len(titles)):
    titles[i]=titles[i].strip() # Suppression de potentielles lignes vides
    #print(titles[i])
## Examen des deux fichiers
print (titles[:10]) # les 10 premiers titres
print (synopses[0][:200]) # les 200 premiers caractères du premier film (premier élément de synopses)
La liste synopses contients les 100 documents et la liste titles contient les 100 titres correspondants.
```

Opérations de pré-traitement sur les texte: Stopwords, stemming, et tokenizing

- Les stop words (mots vides) sont des mots communs dans un texte à tel qu'il est inutile de les indexer ou de les utiliser dans une recherche. En français, des mots vides évidents sont entre autres « le », « la », « de », « du », « ce », etc...

- Le stemming des termes dans un texte consiste à raciniser chaque mot dans le texte. Il s'agit en générale de supprimer les préfixes et les suffixes afin de ramener le texte à sa stricte racine. Ex : stem(frontal)=front ; stem(fishing)=fish. Le stemming permet la détermination des mots clés (généralement utilisés dans les référencements).

-le tokenizing consiste en un découpage du texte en un ensemble de mots, tout en éliminant les ponctuations. On peut découper un texte en phrases (sentence) ou découper en mots(words). Cette dernière est celle qui sera utilisée dans la procédure de text clustering.

Le module NLT fournit des fonctions qui permettent de réaliser ces trois opérations. En cas de nécessité, installer d'abord toutes les fonctions utiles de NLTK.

```
# nltk.download() # Choisir all

# détermination des Stop Words
stopwords = nltk.corpus.stopwords.words('english') # à adapter dans le cas du français : french
# Stemming
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer("english") # à adapter dans le cas du français : french
```

Pour la tokenization, on va définir deux fonctions. Une première qui tokenize le texte et fait du stemming des tokens (des mots). Une seconde fonction qui fait simplement du tokenize sur le texte. Ces deux fonctions sont définies comme suit :

```
def tokenize_and_stem(text):
    tokens = [word for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent)] # tokenize d'abord par phrase, puis par mot pour s'assurer que les ponctuations sont traitées
    filtered_tokens = []
    for token in tokens: # Elimination des tokens ne contenant pas de lettres (p. Ex., tokens uniquement numériques, ponctuation brute)
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    stems = [stemmer.stem(t) for t in filtered_tokens]
    return stems
def tokenize_only(text):
    tokens = [word.lower() for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent)] # tokenize d'abord par phrase, puis par mot pour s'assurer que les ponctuations sont traitées
    filtered_tokens = []
    for token in tokens: # Elimination des tokens ne contenant pas de lettres (p. Ex., tokens uniquement numériques, ponctuation brute)
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    return filtered_tokens
```

On va appliquer ces deux fonctions sur la liste synopses afin de constituer le vocabulaire des documents. On a deux vocabulaires : vocabulaire avec et sans stemming. Ces deux listes se construisent comme suit :

```
totalvocab_stemmed = []

totalvocab_tokenized = []

for i in synopses:
```

```
    allwords_stemmed = tokenize_and_stem(i) #for each item in 'synopses', tokenize/stem

    totalvocab_stemmed.extend(allwords_stemmed) #extend the 'totalvocab_stemmed' list (généralisation de append())


    allwords_tokenized = tokenize_only(i)

    totalvocab_tokenized.extend(allwords_tokenized)
```

On utilise ces deux listes pour former un data frame comme suit :

```
vocab_frame = pd.DataFrame({'words': totalvocab_tokenized}, index = totalvocab_stemmed)
print ('Il y a ' + str(vocab_frame.shape[0]) + ' items dans vocab_frame')
print (vocab_frame.head(n=10)) # affichage des 10 première observations (termes)
```

On note ici qu'il y a des répétition des à la fois dans les stems mais aussi dans les termes compltes. Pour examiner plus en détail ce texte, on peut appliquer les analyses suivantes :

```
#vocab_frame['words'].value_counts() #nombre de répétitions pour chaque valeur
#vocab_frame['words'].nunique() # nombre de valeurs qui sont dupliquées dans la table
#vocab_frame['duplic']=vocab_frame.duplicated() # recherche la duplication sur l'ensemble des variables de la table (renvoie True ou False).
#On peut alors supprimer les valeurs dupliquées si les fréquences ne sont pas utiles pour l'analyse. Dans ce cas, on fait :
#vocab_frame.drop_duplicates() # supprime les duplication sur toute la table
```

Mais ici, on va garder les valeurs comme telles.

Construction de la matrice TF-IDF

Le principe de construction de la matrie TF-IDF se base sur le tableau de ce genre:

Structure de la matrice des fréquences

	terme 1	terme 2	terme 3	…	termec J	
Doc 1	n11	n12	n13	…	N1j	
Doc 2	n21	n22	n23	…	n2j	← Document vector

Doc 3	n31	n32	n33	…	n3j
…	…	…	…	…	…
Doc i	ni1	ni2	ni3	…	nij

Word vector

La matrice fournit la fréquence (nij) de chaque terme dans chaque document (ici chaque synopsis).

NB : Il faut juste noter que comme il s'agit ici des clustering des documents. Les document se présenteront alors en lignes alors que les termes seront en colonne.

La matrice TF-IDF est construite à partir de cette première matrice. Pour la construction de la matrice TF-IDF, on va utiliser la fonction TfidfVectorizer() de sklearn. Les paramètres de cette fonction seront fixés comme suit :

-**max_df**: représente la proportion maximale de documents un terme ne doit pas atteindre pourqu'il soit pris en compte dans le calcul de la matrice tfi-idf. Par exemple, un terme qui apparait dans 90% des documents est probablement un terme qui n'apporte pas beaucoup d'informations au moment du clustering. Ici, on fixe max_df=0,8 (soit 80%).

-**min_idf**: représente le pourcentage minimal de documents dans lequel apparait un terme pour qu'il soit pris en compte dans le calcul de la matrice tf-idf. Ici, on choisit min_idf=0,2 (20%).

-**ngram_range**: On fixe le paramètre ngram_range entre 1 et 3. Ce qui signifie juste qu'on s'interesse uniquement aux unigrams, bigrams et trigrams.

En spécifiant ces paramètres, la matrice tf-idf se calcule comme suit :

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=200000,
                                  min_df=0.2, stop_words='english',
                                  use_idf=True,
tokenizer=tokenize_and_stem, ngram_range=(1,3)) # definition du vectorizer des parameters. ici on utilise la fonction tokenize_and_stem définit plus comme le tokenizer
```

NB: Pour lse stops word en Français, il faut créer en amont une liste de mots avec nltk et assigner cette liste à l'option stop_words=. Pour cela, on peut juste utiliser la variable stopwords créer plus haut en mettant juste la valeur french.

```
tfidf_matrix = tfidf_vectorizer.fit_transform(synopses) # calcul de la tfidf_matrix sur synopses
print(tfidf_matrix.shape)
```

Construction du vocabulaire nettoyé

```
terms = tfidf_vectorizer.get_feature_names() # On récupère juste le termes issus de la construction de la matrie tf-idf.
```

Construction de la matrice de distance

La matrice de distance permet de mesurer la dissimilarité entre chaque paire de document. La distance entre deux documents (deux vecteurs documents) est égale à 1 moins la cosine simularity entre ces deux documents. La fonction ci-dessous permet d'obtenir la matrice de distance.

```
from sklearn.metrics.pairwise import cosine_similarity
dist = 1 - cosine_similarity(tfidf_matrix)
print(dist)
```

La matrice tf-idf et la matrice de distance étant maintenant calculées, on peut appliquer différents algorithmes de clustering. Voir ci-après.

3.16.2.2. Application du K-means clustering à partir de la matrice de distance

A titre illustratif, on va fixer k=5 pour constituer 5 clusters. La mise en œuvre se fait comme suit :

```
from sklearn.cluster import KMeans
num_clusters = 5 # fixe le nombre de clusters
km = KMeans(n_clusters=num_clusters,random_state=2016) # spécification du modèle. On fixe aussi le seed du random generator afin de pouvoir reproduire les résultats à chaque exécution.
km.fit(tfidf_matrix) # Estimation du modèle. Attention, on utilise la matrice tf-idf
clusters = km.labels_.tolist() # génère le valeurs des clusters pour chaque document
```

Les clusters étants crées pour chaque document, on va rassembler les titres, les synopses et les clusters dans un data frame

```
films = { 'title': titles, 'synopsis': synopses, 'cluster': clusters } # création dictionnaire
frame = pd.DataFrame(films, index = [clusters] , columns = ['title', 'cluster']) print(frame['cluster'].value_counts()) #nombre de films par cluster (clusters id allant de 0 à 4)
```

Affichage des mots qui sont plus proches des centroides dans chaque clusters

Après l'application de l'algorithme de clustering, on peut aussi vouloir afficher les n mots les plus proches des centroides de chaque cluster afin de pouvoir qualifier le cluster. Pour cela, on suit les étapes suivantes :

Exemple : affichage des 10 mots les plus proches des centroides et tous les titres de films correspondant à ce cluster

```
 from __future__ import print_function # A mettre au tout début de fichier (plus haut)
print("Les plus proches termes du centroide:")
print()
centroids_coord = km.cluster_centers_# récupération des coordonnés des termes par rapport au centroide
order_centroids = centroids_coord.argsort() #tri des termes (mots) selon leur proximité au centroide ( le tri est  croissant). Attention la fonction argsort() renvoie les indices
order_centroids = order_centroids[:, ::-1] # réorganise la liste du dernier au premier élément (objectif: tri décroissant des proximités)
for i in range(num_clusters):
    print("Cluster %d:" % i, end='')
    for ind in order_centroids[i, :10]: # Récupère les 10 premiers éléments de order_centroids trié. Remplacer n par le nombre de mots souhaité par cluster (ind se refère aux indices renvoyés par argsort())
    #print('       %s'       %       vocab_frame.ix[terms[ind].split(' ')].values.tolist()[0][0].encode('utf-8', 'ignore'), end=',')
        termLab=terms[ind] # récupération du terme correspondant à ind dans dans terms (le vocabulaire nettoyé)
        termLab=termLab.split(' ') # pour mettre terme sous forme de list avec quote, utilisable dans l'indiçage
        termVal=vocab_frame.ix[termLab].values # récupère les valeurs pour lesquelles les indices dans le data frame vocab_frame sont égaux à termLab (forme array)
        termValList=termVal.tolist()  #  convertit  l'array  en  list (liste de liste)
        termValListFirst=termValList[0][0]  #  Récupère  le  premier élément de la liste des listes. Le second [0] est mis pour récupérer le string sans les crochets

      #Notons ici qu'on a choisit le premier mot de chaque liste comme mot représentatif. On pouvait choisir un autre autre ordre. Ex: deuxième mot [1][0] on obtiendrait une autre variante du mot : ex : de kill on on a killed, etc…
        print(' %s' % termValListFirst, end=',')
    print() #Espace vide
    print() # Espace vide
    print("Cluster %d titles:" % i, end='')
```

```
    for  title  in  frame.ix[i]['title'].values.tolist #  sélection  de
toute  les  lignes  dont  l'index  equivaut  à  i,  ensuite  sélection  de
valeurs de titles et enfin conversion en list
        print(' %s,' % title, end='') # fais le print sur la même
ligne avec virgule comme séparateur.
        #print(' %s,' % title) # print sur des lignes séparées
    print()# Espace vide
    print()# Espace vide

print()
print()
```

NB : Pour plus de détails sur la mise en œuvre du K-means clustering, se référer au chapitre 7.

3.16.2.3. Application de la Classification Ascendente Hierarchique (CAH) à partir de la matrice de distance

La CAH ou le Hierarchical clustering est une méthode de classification dont le but est de répartir les individus dans un certain nombre de classes. La méthode suppose qu'on dispose d'une mesure de dissimilarité entre les individus. La classification est dite ascendante car elle part d'une situation où tous les individus sont seuls dans une classe, puis sont rassemblés en classes de taille de plus en plus grandes.

D'une manière générale, la méthode se met en œuvre en deux temps. Dans un premier temps, on construit la matrice de linkage en utilisant une méthode de calcul. Dans cette phase, on peut choisir des méthode de linkage comme celles de Ward. Cette matrice permet de représenter le dendrogram. L'analyse du dendrogram permet ensuite de se faire une idée sur le nombre de clusters à retenir.

En utilisant la matrice de distance calculée précédemment avec la matrice tf-idf sur les documents, la CAH se met en œuvre comme suit :

```
from scipy.cluster.hierarchy import ward, dendrogram
linkage_matrix = ward(dist) #Définition de la matrice de linkage
# représentation du dengrammme
fig, ax = plt.subplots(figsize=(15, 20)) # set size
ax        =        dendrogram(linkage_matrix,        orientation="right",
labels=titles,color_threshold=0) # dans découpage
#plt.tick_params(\     axis= 'x',       which='both',      bottom='off',
top='off',     labelbottom='off')
#plt.tight_layout() #affiche le graph avec  tight layout (agrandit la
zone de traçage)
plt.show()
```

Au vu du dendogramme, on va choisir le nombre de clusters égal à 5 en modifiant la valeur color_threshold. L'option color_threshold permet de choisir le nombre de clusters. La valeur de color_threshold se fixe en se référant à la graduation de l'axe. On choisit la valeur au niveau de laquelle on va couper l'arbre pour former les clusters. Par exemple, en choisissant ici color_threshold=3, on obttient 4 clusters. Et avec color_threshold=2.5, on obtient 5 clusters.

```
# ici on choisit  color_threshold=2.5. Ainsi, on a :
ax       =       dendrogram(linkage_matrix,       orientation="right",
labels=titles,color_threshold=2.5) # Trace un nouveau dendro
plt.show()
plt.savefig('ward_clusters.png', dpi=200) #enregistrement du dendro
# Création de la variable des 5 clusters retenus ( découpage effectué
au niveau 2.5)
import scipy.cluster.hierarchy as cah
clusters_cah = cah.fcluster(linkage_matrix,t=2.5,criterion='distance')
# Découpage au niveau t=2.5
print(clusters_cah) # les clusters
# Création du data frame associant les titres des films et les
clusters
df                                                                  =
pd.DataFrame(dict(cluster_kmean=clusters,cluster_cah=clusters_cah,
title=titles)) # crée un data frame conteant les clusters obtenus par
cah. On intègre aussi les clusters obtenus par k-mean
print(df)
```

NB : Pour plus de détails sur la mise en œuvre de la classification hiérarchique ascendante, se référer au chapitre 7.

3.16.2.4. Application du Multidimensional scaling (MDS) à partir de la matrice de distance

Le Multidimensional scaling (en français, le positionnement multidimensionnel) encore appelé MDS est un ensemble de techniques utilisées pour explorer les similarités et les dissimilarités dans les données. Pour sa mise en œuvre, on part d'une matrice de similarité (matrice de distance) entre les individus, puis affecte une position à chaque individu dans un espace à N dimensions, où 'N' est précisé a priori. Par exemple, pour N=2, on a un espace à deux dimension et pour N=3, les positions peuvent être représentées à l'aide d'un graphe ou en 3D.

Avec la matrice de distance précédemment calculée avec la fonction cosine_similarity() à partir de la matrice tf-idf, on va construire un plan à deux dimensions basé sur les deux premières composantes en utilisant le MDS. Par la suite on va faire une représentation graphique des documents dans le plans formé par ces deux composantes par MDS. A titre illustratif, on distinguera les individus par des couleurs basées sur les clusters précédemment obtenus par K-means clustering. Ces étapes sont mises en œuvre comme suit :

```
from sklearn.manifold import MDS
MDS()
mds      =      MDS(n_components=2,      dissimilarity="precomputed",
random_state=2016) # définition du modèle. On retient 2 composantes.On
fixe le random seed à 2016 pour la reproductibilité.
pos = mds.fit_transform(dist)  # application du modèle
xs= pos[:, 0] # l'axe des x (composante 1)
ys = pos[:, 1] # l'axe des y (composante 2)
print()
print()
```

Représentation des documents

On va représenter les individus dans les deux axes factoriels constitués des deux composantes. Toutefois pour mieux visualiser les individus, on va les labéliser avec les clusters déjà obtenus avec K-means et ainsi attribuer des couleurs aux individus selon les clusters. Il faut noter que la labélisation pouvait aussi se faire avec une autre variable catégorielle dispobible (pas nécessairement les clusters). Dès lors la combinaison des résultats du MDS et des clusters du K-Means se met en oeuvre comme suit:

```
df  =  pd.DataFrame(dict(x=xs,  y=ys,  label=clusters,  title=titles))  #
crée un data frame à partir de xs, ys, cluster et titles
import matplotlib.pyplot as plt # pour la représentation graphique
import matplotlib as mpl # pour la représentation graphique
cluster_colors  =  {0:  '#1b9e77',  1:  '#d95f02',  2:  '#7570b3',  3:
'#e7298a', 4: '#66a61e'} # choix de couleur des clusters
cluster_names = {0: 'Family, home, war', 1: 'Police, killed, murders',
2: 'Father, New York, brothers', 3: 'Dance, singing, love',
                 4:  'Killed,  soldiers,  captain'}  #  choix  du  libellé
des clusters

fig, ax = plt.subplots(figsize=(17, 9)) # cadre du graphique
ax.margins(0.05) #  Optiennel, ajoute  5% au dimensions des échelles
groups  =  df.groupby('label')  #  scinde  la  table  selon  les  valeurs  de
labels
for name, group in groups: # Récupère les valeurs des groupes-cluster
(names) et les portions de données par groupe (group). La boucle sert
à représenter chaque cluster dans le graph(même cadran)
    ax.plot(group['x'], group['y'], marker='o', linestyle='', ms=12,
            label=cluster_names[name],      color=cluster_colors[name],
mec='none')
    ax.set_aspect('auto')
    ax.tick_params(\
```

```
        axis= 'x',
        which='both',
        bottom='off',
        top='off',
        labelbottom='off')
    ax.tick_params(\
        axis= 'y',
        which='both',
        left='off',
        top='off',
        labelleft='off')

ax.legend(numpoints=1)  #Légende représentée par un seul point
for i in range(len(df)):
    ax.text(df.ix[i]['x'],  df.ix[i]['y'],  df.ix[i]['title'],  size=8)
# jouter les labels aux points avec les titres des films

plt.show() #affiche le graph
plt.close()
```

NB: Ce graphique peut être amélioré en utilisant le module mpld3 qui offre beaucoup plus de fonctionnalité en plus du matplotlib.

3.17. Stratégie générale de pré-traitement de texte en vue de la recherche d'information, du text matching ou du text clustering

Dans la section précédente, nous avons présenté quelques méthodes visant à extraire des informations à partir d'un ensemble de textes notamment le text matching et le text clustering afin de réaliser des extraction d'informations. Cette section vise à présenter simplement quelques étapes clées qu'il faut avoir à l'esprit lors du traitement de texte. Ces étapes peuvent être résumées comme suit :

- D'abord, après avoir importer les documents sous forme de texte en format string, la première chose à faire est de découper le texte en des tokens de mots en utilisant successivement les fonctions sent_tokenize() et word_tokenize() de NLTK afin d'éliminer les ponctuations avec une plus grande efficacité.

- Ensuite, le text entier se présentant maintenant sous forme de liste de termes, il faut penser à éliminer les stopwords (en servant du corpus de langue étudiée). Voir section concernant les stopword.

- Dans un troisième temps, penser à lemmatiser les termes restants afin de réduire les termes à leur stricte composante (plus informative)

- Enfin pour améliorer les résultats de la lemmatisation, on peut appliquer le stemming sur les lemmes afin de les réduires à leur racine. Une telle transformations peut s'avérer utile dans le recherche de mots clés ou dans le texte matching ou text clusterings

Chapitre 4 : Traitement et Organisation des tables de données

Ce chapitre a pour but de présenter les opérations courantes de traitements et d'organisations des données sous Python. Les bases de données se présentent sous le format data frame. Nous nous servirons essentiellement du module pandas qui est spécialisé dans la manipulation des objets data frame.

Signalons toutefois que le data frame est constitués d'un ensemble de colonnes dont chacun peut être considéré comme une série. La série est en fait un data frame à une seule colonne. Les sections ci-dessous donnent plus de détails sur la création et la manipulation des objets series et data frames.

4.1. Création des objets « series »

Un objet series est un objet python qui se présente sous la forme d'un vecteur-colonne. C'est un objet spécifique au module pandas. On distingue deux princiaples méthodes pour créer un objet series : la création de série à partir d'une liste de valeurs ou la création de séries à partir des dictionnaires ; Ci-dessous la mise en application de ces deux méthodes :

4.1.1. Création d'une série à partir d'une liste de valeurs

On peut créer une série par saisie (ou à partir d'une liste) en utilisant la fonction series() de pandas. Exemple:

```
import pandas
mySerie1 = pandas.Series([5, 6, 2, 9, 12])
print(mySerie1)
myList=[5, 6, 2, 9, 12])
mySerie2 = pandas.Series(myList)
print(mySerie2)
```

4.1.2. Création d'une série à partir d'un dictionnaire

Par définition, un dictionnaire est un ensemble formé d'une valeur et d'un indice. Etant donné cette structure, on peut définir une série en utilisant directement le dictionnaire sans avoir besoin de distinguer d'une part les valeurs et d'autre part les indices comme pour le cas d'un liste. Exemple: soit le dictionnaire suivant:

```
myDico = {'Cochise County': 12, 'Pima County': 342, 'Santa Cruz
County': 13, 'Maricopa County': 42, 'Yuma County' : 52}
On peut définir une série à partir de ce dictionnaire comme suit:
mySerie = pandas.Series(myDico)
print(mySerie)
```

En convertissant un dictionnaire en series, les indices du dictionnaire deviennent directement les indices de la série.

4.1.3. Définir des indices pour les séries (identifiant des séries)

Par défaut les séries sont indexé de 0 , 1, ...,n. On peut alors ajouter des indices aux séries en utilisant l'option index lors de la création de la série. Ex:

```
mySerie = pandas.Series([5, 6, 2, 9, 12],index=['Cochise County',
'Pima County', 'Santa Cruz County', 'Maricopa County', 'Yuma County'])
print(mySerie)
```

On remarque que les séries et leur indice sont tous définis sous forme de liste.

NB: Lorsque les valeurs proviennent de la défintion d'une liste, on peut écrire la définition de la série comme suit:

```
myList=[5, 6, 2, 9, 12] # définition de la liste des valeurs
myIndices=['Cochise County', 'Pima County', 'Santa Cruz County',
'Maricopa County', 'Yuma County'] # définition de la liste des indices
mySerie = pandas.Series(myList, myIndices) # définition de la série
print(mySerie)
```

On peut ainsi remplacer les indices dans les opérations pour accéder aux valeurs. Exemple :

```
print(mySerie[0]) # Selection du premier élément (méthode classique)
print(mySerie["Cochise County"]) # Selection du premier élément
(sélection par l'index)
```

Redéfinition des indices d'une série:

Pour rédéfinir les indices d'une série, on utilise la fonction reindex(). Exemple:

```
mySerie = mySerie.reindex(['Tombstone', 'Douglas', 'Bisbee', 'Sierra
Vista', 'Barley'])
```

La fonction reindex() remplace l'ancien index par un nouveau.

Bien entendu, on peut appliquer les fonctions reindex, si l'on souhaite après modifier les indices.

4.2. Création de table de données (data frame)

Tout comme les objets series, les objets data frames sont des objets python spécifiques au modules pandas. Il existe plusieurs manières de créer les objets data frames : création à partir de series, création à partir de dictionnaires, création à partir de sources de données externes, etc... La fonction de base pour créer un data frame est la fonction DataFrame() de pandas.

4.2.1. Création de data frame à partir des séries

Exemple: Soient les 15 séries suivantes définies comme suit:

```
year = pandas.Series([1875, 1876, 1877, 1878, 1879, 1880, 1881, 1882,
1883, 1884,1885, 1886, 1887, 1888, 1889, 1890, 1891, 1892, 1893,
1894])
guardCorps = pandas.Series([0,2,2,1,0,0,1,1,0,3,0,2,1,0,0,1,0,1,0,1])
corps1 = pandas.Series([0,0,0,2,0,3,0,2,0,0,0,1,1,1,0,2,0,3,1,0])
corps2 = pandas.Series([0,0,0,2,0,2,0,0,1,1,0,0,2,1,1,0,0,2,0,0])
corps3 = pandas.Series([0,0,0,1,1,1,2,0,2,0,0,0,1,0,1,2,1,0,0,0])
corps4 = pandas.Series([0,1,0,1,1,1,1,0,0,0,0,1,0,0,0,0,1,1,0,0])
corps5 = pandas.Series([0,0,0,0,2,1,0,0,1,0,0,1,0,1,1,1,1,1,1,0])
corps6 = pandas.Series([0,0,1,0,2,0,0,1,2,0,1,1,3,1,1,1,0,3,0,0])
corps7 = pandas.Series([1,0,1,0,0,0,1,0,1,1,0,0,2,0,0,2,1,0,2,0])
corps8 = pandas.Series([1,0,0,0,1,0,0,1,0,0,0,0,1,0,0,0,1,1,0,1])
corps9 = pandas.Series([0,0,0,0,0,2,1,1,1,0,2,1,1,0,1,2,0,1,0,0])
corps10 = pandas.Series([0,0,1,1,0,1,0,2,0,2,0,0,0,0,2,1,3,0,1,1])
corps11 = pandas.Series([0,0,0,0,2,4,0,1,3,0,1,1,1,1,2,1,3,1,3,1])
corps14 = pandas.Series([ 1,1,2,1,1,3,0,4,0,1,0,3,2,1,0,2,1,1,0,0])
corps15 = pandas.Series([0,1,0,0,0,0,0,1,0,1,1,0,0,0,2,2,0,0,0,0])
```

Pour rassembler ces séries en dataframe, on va passer par deux étapes intermédiaires notamment la création de dictionnaires

Premièrement: Rassembler les séries dans un dictionnaire en utilisant la fonction dict() pour créer les variables. On a:

```
variables = dict(year=year, guardCorps = guardCorps, corps1 = corps1,
                 corps2 = corps2, corps3 = corps3, corps4 = corps4,
                 corps5 = corps5, corps6 = corps6, corps7 = corps7,
                 corps8 = corps8, corps9 = corps9, corps10 = corps10,
                 corps11 = corps11 , corps14 = corps14, corps15 =
corps15)
```

Deuxièmement: On crée le dataframe en utilisant ma fonction DataFrame de pandas.

```
mydata = pandas.DataFrame(variables, columns = ['year', 'guardCorps',
'corps1', 'corps2','corps3', 'corps4',
                                              'corps5',
'corps6','corps7', 'corps8','corps9', 'corps10',
                                              'corps11',
'corps14','corps15'])
```

NB: Ici, la définition columns n'est pas nécessaire car les variables ont les mêmes noms que ceux du dictionnaire. Elle sert juste à garder l'ordre d'appariation des variables dans un print.

4.2.2. Création du dataframe à partir d'un dictionnaire

On peut créer un dataframe à partir d'un dictionnaire en utilisant la fonction DataFrame de pandas.

Exemple: Soit le dictionnaire suivant:

```
dico = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'last_name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze'],
        'age': [42, 52, 36, 24, 73],
        'preTestScore': [4, 24, 31, 2, 3],
        'postTestScore': [25, 94, 57, 62, 70]}
```

On crée le dataframe comme suit:

```
mydata = pandas.DataFrame(dico, columns = ['first_name', 'last_name',
'age', 'preTestScore', 'postTestScore'])
print(mydata)
```

4.2.3. Création d'un dataframe vide (d'observations)

Un dataframe vide est un dataframe dont les variables (colonnes) sont indiquées mais qui ne contient pas d'observations (lignes). Le plus souvent, on a besoin de créer un data frame afin de l'ajouter des observations dans la suite du programme en utilisant les fonctions appends. Voic-ci-dessous un exemple simple de création de data frame vide.

mydata = pandas.DataFrame(index=range(0), columns = ['id_var', 'myvar1', ' myvar2'])

l'argument index permet d'indiquer que le dataframe ne contient aucune ligne. Et l'argument columns permet d'indiquer les noms des variables.

NB : Pour ajouter des observations à ce data frame vide, on utilise les fonction de fusions verticales (voir la section de fusions de tables de données).

4.3. Création de dataframe à partir des sources de données externes

Les données externes peuvent se présenter sous plusieurs formes: csv, txt, xlsx, etc... Cette section a pour but de décrire comment importer les données en python sous forme de data frame.

4.3.1. Importation des fichiers txt et csv

L'importation de fichiers txt ou csv se fait basiquement avec la fonction read_csv() du module pandas (on peut aussi utiliser la fonction from_csv()).

Importation de fichiers txt avec séparateur « tabulation »

```
x=pandas.read_csv("mytabdelimdata.txt",sep="\t",      header=0,encoding
='utf-8') # ou encoding='Latin-1' pour les tables avec caractères
accentués
print(x)
```

Importation de fichiers csv avec séparateur « ; »

y=pandas.read_csv("mycsvdata.csv",sep=';', header=0, encoding ='utf-8')

print(y)

NB : Dans les deux exemples ci-dessus, le paramètre header=0 indique que les noms de variables se trouvent sur la ligne 0. Si les données ne contiennent pas les données, on met header=None.

On peut appliquer cette méthode d'importation pour tous les fichiers txt ou csv. Il suffit de changer le séparateur sep (Ex : ;, , ,etc…).

Penser aussi à d'autres types d'encodages. L'encodage doit être choisi en fonction du type d'encodage du fichier. Si la table contient des caractères accentués on peut utiliser encoding ='Latin-1'. Penser également à utiliser l'option skiprows=1 pour ignorer certaines lignes.

Ajout des noms des colonnes (après importation)

Lorsque la table de données importée ne contitent pas les noms des colonnes, on ajoute les noms dans une seconde étape. Ex:

```
x.columns=['nomVar1', 'nomVar2', 'nomVar3', 'nomVar4', 'nomVar5'] #
ajout des noms des colonnes
```

Gestion des valeurs manquantes lors de l'importation de fichiers csv ou txt

Les fichiers à importer peuvent contenir des valeurs manquantes. Ces valeurs manquantes se présentent sous différentes formes (espace vide entre deux délimiteurs ou une autre valeur comme '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'nan'.

Les valeurs de type « NA » et ses différentes variantes sont par défaut reconnues par pandas lors de l'importation. En revanche les valeurs en « espace vide » nécessitent d'utiliser des options supplémentaires. Les exemples ci-dessous fournissent des détails sur la gestion des valeurs manquantes lors de l'importation.

Cas où la valeur manquante est représentée par un espace vide (blank)

Dans ce cas, il faut utiliser l'option « **skipinitialspace=True** » comme suit :

```
mydata=pandas.read_csv("myfile_with_space.csv",sep=',',
header=0,skipinitialspace=True, encoding ='utf-8')
```

```
print(mydata.iloc[210:250]) # visionner quelques lignes contenant les valeurs manquantes
```

En utilisant cette option, pandas remplace ces cellules vides par des valeurs NaN reconnues comme des valeurs manquantes en tant que telles. On peut alors effectuer toutes les opérations de manipulation sur les variables contenant ces valeurs (opération arithmétiques : addition, soustraction, etc..).

NB : Si on n'utilise pas l'option skipinitialspace=True, les valeurs manquantes resteront des cellules vides dans la table importée. Dans ce cas, il n'est plus possible de faire des opérations arithmétiques sur les variables qui les contiennent(addition, soustraction, etc..). Car la variable importée se présente sous format caractère. Cette situation convient uniquement lorsque la cellule vide est une valeur en soi.

Cas où la valeur manquante est représentée par une valeur de type NA ou une de ses variantes (section à revoir et à confirmer)

Lorsque la valeur manquante est représentée par l'une des valeurs suivantes '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'nan', python les reconnait directement sans qu'on ait besoin de spécifier une option supplémentaire. Ex : fichier avec "NA"

```
mydata2=pandas.read_csv("myfile_with_NA.csv",sep=',',        header=0, encoding ='utf-8')
print(mydata2.iloc[210:250]) # visionner quelques lignes contenant les valeurs manquantes
```

Cas où la valeur manquante est représentée par une valeur spéciale (ex : -999)

Lorsque la valeur manquante est représentée par une valeur spéciale (ex : -99, -999 ,etc), il convient d'abord d'importer les données avec la méthode de base. Ensuite remplacer cette valeur spéciale avec une valeur NaN en utilisant la fonction replace. Exemple :

```
mydata=pandas.read_csv("mycsvdata.csv",sep=';',   header=0,   encoding ='utf-8') # importation
import numpy
mydata=mydata.replace(-999, numpy.nan) # Remplace -999 par une valeur NaN reconnaissable comme une valeur manquante en tant que telle.
```

NB : Lorsque la valeur manquante est représentée par une valeur spéciale, on peut aussi utiliser l'option « na_value » pour spécifier cette valeur. Consulter la description complète de la fonction read_csv() à ce lien : http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

*Cas où la valeur manquante est une valeur quelconque : utilisation de l'option **na_values***

Lorsque la valeur manquante est une valeur quelconque, on peut utiliser l'option « na_values ». Exemple : Soit le fichier csv nommé mycsvfile construit comme suit :

```
df = pandas.DataFrame({'A' : [-999, 2, 3], 'B' : [1.2, -999.0, 4.5]}) # Création des données
df.pandas.DataFrame.to_csv('mycsvfile.csv', sep=' ', index=False) # Création du fichier csv avec séparateur espace.
```

Cherchons maintenant à importer ce fichier en utilisant plusieurs types de valeurs manquantes :-999, -999.0. Ci-dessous des exemples :

```
mydata= pandas.DataFrame.read_csv('mycsvfile.csv', sep= ' ', header=0, na_values=[-999]) # Traite uniquement la colonne A
mydata= pandas.DataFrame.read_csv('mycsvfile.csv', sep= ' ', header=0, na_values=[-999.0]) # Traite uniquement la colonne B
```

NB : On peut utiliser des valeurs en caractères dans l'option na_values. Ex : na_values=['ms'].

4.3.2. Importation de fichiers Excels

L'importation de data frames à partir de fichiers excel se fait avec la fonction read_excel() de pandas. On dispose pour cela de plusieurs méthodes. Exemples :

Première méthode

```
z1=pandas.read_excel("myxlsxdata.xlsx",sheetname="myxlsxdatasheet",
header=0, parse_cols="A:W")
print(z1)
```

Lorsque le fichier ne contient pas le nom des variables, on fait header=None.

Ensuite on définit les noms des colonnnes en faisant :

```
z1.columns=['name1', 'name2', 'name3', 'name4', 'name5'] # ajout des noms des colonnes
```

Deuxième méthode

```
fichier = pandas.ExcelFile('myxlsxdata.xlsx') # lecture du document myxlsxdata.xlsx
z2 = fichier.parse('myxlsxdatasheet') # lecture parsing de la feuile de données myxlsxdatasheet
print(z2)
```

4.3.3. Importation de données à partir des sources de données non structurées

Souvent les données à importer se présentent sous des formes non structurées. Par exemple un fichier de texte, une page html, un ensemble de caractères à partir duquel il faut extraire les

données. Dans ces cas, on est obligé d'abard de passer par des opérations de traitement de texte pour constituer les données. L'utilisation des expressions régulières peut également s'avérer nécessaire. Dans cette section, nous donnons deux exemples de cas où ces situations se présentent:

4.3.3.1. Exemple 1 : Convertir un ensemble de texte en un dataframe avec des méthodes classiques de traitement de texte

Pour convertir un ensemble de texte en un dataframe, on se sert généralement des méthodes de traitement de texte. Les données suivantes écrites sous forme de texte (qui donne la consommation d'alcool et de tabac en grande Brétagne par région:

```
mytext = '''Region Alcohol Tobacco
North 6.47 4.03
Yorkshire 6.13 3.76
Northeast 6.19 3.77
East Midlands 4.89 3.34
West Midlands 5.63 3.47
East Anglia 4.52 2.92
Southeast 5.89 3.20
Southwest 4.79 2.71
Wales 5.27 3.53
Scotland 6.08 4.51
Northern Ireland 4.02 4.56'''
```

En observant ce texte, on peut distinguer trois colonnes dans ce texte: région, Alcohol et Tobacco. Pour transformer ce texte en dataframe, on va suivre les étapes suivantes:

1- On découpe le texte en liste où chaque élément correspond à une ligne

```
d = mytext.split('\n')
print(d)
```

L'opérateur '\n' sert à indiquer que le séparateur ici est la ligne et l'espace. Sinon la liste serait constitué de chaque mot et non chaque ligne. Chaque élément de la liste est composée de trois valeurs : valeur région, valeur alcool et valeur tabac.

2- On va découper ces trois élements en des listes (on formera alors une liste des listes)

```
d = [ i.split(' ') for i in d ]
print(d) # on a une liste des listes
```

Découper les valeurs en des listes (les textes sont séparés par les espaces ' '. Cela pouvait aussi être une tabulation '\t' ou un espace de type '\s'

3- Transformer les valeurs du format caractère en format numérique pour les données Alcohol et Tobacco

```
for i in range(len(d) ): # la longueur de d est égal au nombre de sous listes.
    for j in range( len( d[0] ) ):        try:
            d[i][j] = float( d[i][j] )
        except:
            pass
```

Comme toutes les sous-listes ont la même longueur, on garde uniquement celle de la première. On pouvait aussi indiquer simplement 3.

4- Construction du dataframe

```
df = pandas.DataFrame( d[1:], columns=d[0] ) # Ici les données commence à partir de l'indice 1 et les noms des variables sont à l'indice 0
print(df)
```

4.3.3.2. Exemple 2 : Convertir un ensemble de texte en dataframe en utilisant les expressions régulières (regex)

On peut aussi utiliser les expressions régulières pour extraire les valeurs des variables et reconstituer le dataframe.

Soit le texte défini par le dictionnaire suivant.

```
textData = {'rawText': ['Arizona 1 2014-12-23       3242.0',
                'Iowa 1 2010-02-23       3453.7',
                'Oregon 0 2014-06-20       2123.0',
                'Maryland 0 2014-03-14       1123.6',
                'Florida 1 2013-01-15       2134.0',
                'Georgia 0 2012-07-14       2345.6']}
```

Importation du texte

```
df = pandas.DataFrame(textData , columns = ['rawText']) # dataframe à une seule colonne
print(df)
```

Comme on peut le constater, ce texte regorge quatre variables dont les valeurs ont été collées. On peut distinguer les états (Arizona, Iowa,...), le sexe (1 et 0), la date (ex: 2014-12-23), les espaces et une variable numérique (score). Il va donc falloir décomposer cette chaîne en plusieurs variables. Pour cela, on va s'appuyer sur les expressions régulières regex. La démarche sera la suivante :

Considérons qu'il y ait quatre variables: state, sex, date et score. Extrayons chacune de ces variables. Etant donné que chaque variable se présente sous un format propre à elle, on peut élaborer des expressions suivantes pour extraire les valeurs:

Extraction de la variable state

```
df['state'] = df['raw'].str.extract('([A-Z]\w{0,})') # analyse l'opéartion regex
```

Extraction de la variable sex(female=1)

```
df['female'] = df['raw'].str.extract('(\d)') # female est une variable à 1 chiffre
```

Extraction de la date

```
df['date'] = df['raw'].str.extract('(....-..-..)')
```

La date est sous le format 4 caractères (de toute sorte)+ un tiret + 2 caractère+ 1 tiret+ 2 caractères (ici les chiffres sont traités comme caractères)

Extraction de la variable score

df['score'] = df['raw'].str.extract('(\d\d\d\d\.\d)') # score est une variable à 4 chiffres +.+ 1 chiffre

print(df)

4.4. Exportation des données vers des formats externes

On peut exporter les data frames vers de formats externes commes les fichiers csv ou excel. Pour cela, on utilise la fonction to_csv() ou to_excel() de pandas. Exemples :

4.4.1. Exportation vers le format csv :

mydata.to_csv("mydata_exp1.csv",sep=';',header=True) # avec ';'

mydata.to_csv("mydata_exp2.csv",sep=',',header=True) # avec ','

4.4.2. Exportation vers le format Excel (utilisation de l'engine xlswriter

NB : avant d'utiliser ExcelWriter de pandas, assurez-vous que les modules complémentaires sont installés : openpyxl et XlsxWriter qui sont utiles pour le bon fonctionnement de l'importation et l'exportation

```
import pandas
#import openpyxl
#import XlsxWriter
mywriter = pandas.ExcelWriter('mydata_exp.xlsx' , engine='xlsxwriter')
# definition du fichier excel
mydata.to_excel(mywriter, sheet_name='data', header=True, index=False, startrow=0, startcol=0)
```

```
mywriter.save() # stocker le résultat
```

Noter aussi qu'on peut stocker plusieurs dataframse dans un même fichier excel. Pour cela, il faut exécuter la fonction to_excel sur chaque dataframe avant de faire save(). Exemple :

```
mywriter = pandas.ExcelWriter('output.xlsx', engine='xlsxwriter')
mydata1.to_excel(mywriter, 'data1') # Enregistre dans la feuille data1
mydata2.to_excel(mywriter, 'data2') # Enregistre dans la feuille data2
writer.save() # enregistre le fichier excel en output.
```

En général, les modules utiles pour la gestion et la manipulation des fichiers excels sont : openpyxl, XlsxWriter, xlrd, xlwt et xlutils. Il est recommandé d'installer une fois pour toutes ces modules avant toute opération impliquant les fichiers excels.

Par ailleurs, il arrive qu'on veuille faire append à un fichier excel existant. Dans ce cas, on va suivre l'exemple suivant en utilisant l'engine openpyxl:

```
import pandas
from openpyxl import load_workbook
book = load_workbook('myExcelfile.xlsx')
writer = pandas.ExcelWriter('myExcelfile.xlsx', engine='openpyxl')
writer.book = book
writer.sheets = dict((ws.title, ws) for ws in book.worksheets)
data_filtered.to_excel(writer, "Main" )
writer.save()
```

On peut aussi effectuer des manipulations des contenus de fichiers excels (ouverture, copie de valeurs, etc…) en utilisant les modules xlrd, xlwt et xlutils.copy. Exemple : On dispose de 7 fichiers excels contenant des données et un fichier supplémentaire qui contient la description des colonnes. On souhaite copier le contenu de fichiers en ajoutant une nouvelle feuille à chacun des 7 fichiers. Pour cela, on va suivre la démarche suivante :

```
import xlrd
import xlwt
from xlutils.copy import copy
import os
if not os.path.exists("/new"): os.makedirs("new") # Crée un dossier
nommé New au cas il n'existerait pas.
toBeAppended = xlrd.open_workbook("ToBeAppended.xlsx")
sheetToAppend = toBeAppended.sheets()[0]      #Sélectionne la première
feuille
dataTuples = [] # Crée une liste vide
for row in range(sheetToAppend.nrows):
    for col in range(sheetToAppend.ncols):
        dataTuples.append((row,                                    col,
sheetToAppend.cell(row,col).value))   # ajoute le numéro de ligne, de
colonne ainsi que la valeur de la cellule au tuple
```

```
wbNames = ["{}.xlsx".format(num) for num in range(1,7)] # Crée les noms des 7 fichiers
for name in wbNames:
    wb = copy(xlrd.open_workbook(name))
    newSheet = wb.add_sheet("Appended Sheet") # ajout nouvelle feuille
    for row, col, data in dataTuples:
        newSheet.write(row, col, data) # colle la valeur de la cellule
    wb.save("new/"+name.split('.')[0]+".xls")  # L'enregistrement est possible uniquement en xls
```

Exemple 2 : Créer une feuille excel et ajouter des données sur une nouvelle feuille (utilisation du module xlwt)

```
import xlwt

x=1
y=2
z=3

list1=[2.34,4.346,4.234]

book = xlwt.Workbook(encoding="utf-8")

sheet1 = book.add_sheet("Sheet 1")

sheet1.write(0, 0, "Display")
sheet1.write(1, 0, "Dominance")
sheet1.write(2, 0, "Test")

sheet1.write(0, 1, x)
sheet1.write(1, 1, y)
sheet1.write(2, 1, z)

sheet1.write(4, 0, "Stimulus Time")
sheet1.write(4, 1, "Reaction Time")
i=4

for n in list1:
    i = i+1
    sheet1.write(i, 0, n)
book.save("trial.xls")
```

4.5. Description de la table de données

Soit la table de données suivante:

```
myData=pandas.read_excel("myxlsxdata.xlsx",sheetname="myxlsxdatasheet"
, header=0, parse_cols="A:W")
```

Pour décrire le contenu de la tabme, on utilise les opérations suivantes:

```
print(myData) # Imprime toute la table (toutes les variables et les observation)
print(myData.head(n=5)) # pour afficher les première lignes.
print(myData.tail(n=5)) # pour afficher les dernières lignes.
print(myData.shape) # nombre de lignes et de colonnes
print(myData.columns) # liste des variables
print(myData.dtypes) #liste des colonnes et leurs types
```

4.6. Paramétrer le nombre de colonnes et le nombre de lignes à afficher lors de l'affichage de la table de données

```
pandas.set_option('expand_frame_repr', False) # augmente le nombre de variable par page en faisant print(myData)
pandas.set_option('display.max_columns', 1000) # Fixe  le nombre de colonnes à afficher à 1000
pandas.set_option('display.max_row', 1000000) # Augmente le nombre de lignes à afficher
```

4.7. Opérations sur les variables

4.7.1. Sélectionner les variables

Soit la table:

```
dico = {'name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [4, 24, 31, 2, 3],
        'coverage': [25, 94, 57, 62, 70]}
mydata = pandas.DataFrame(dico, index = ['Cochice', 'Pima', 'Santa Cruz', 'Maricopa', 'Yuma'])
```

4.7.1.1. Sélectionner une seule colonne

Exemple : sélectionner la variable name

```
mydata['name']
print(mydata['name'])
```

4.7.1.2. Sélectionner plusieurs colonnes

```
mydata[['name', 'reports']] # sélectionne les colonnes name er report
mydata.iloc[:,:2] # Sélectionner les colonnes par leur numéro (ex: les deux premières colonnes 0 et 1 index 2 exclus)
mydata.iloc[:, 2:] # Sélection à partir de la troisième colonne : index 2 inclu (correspondant à la troisième colonne)
```

4.7.2. Renommer une colonne dans un dataframe

Soit la table:

```
data = {'Commander': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'Date':  ['2012,  02,  08',  '2012,  02,  08',  '2012,  02,  08',
'2012, 02, 08', '2012, 02, 08'],
        'Score': [4, 24, 31, 2, 3]}
df = pandas.DataFrame(data, index = ['Cochice', 'Pima', 'Santa Cruz',
'Maricopa', 'Yuma'])
print(df)
```

Renommer la variable Commander en Leader, Date en period

```
df.rename(columns={'Commander':       'Leader','Date':       'Period'},
inplace=True)
```

Attention il ne faut pas créer un nouveau objet. Sinon, il faut enlever inplace=True (False). Ainsi, on aura:

```
df2=df.rename(columns={'Commander':     'Leader','Date':     'Period'},
inplace=False)
print(df)
```

4.7.3. Transformer les noms des variables en miniscule ou en majuscule

Exemple:

```
mydata.columns  =  map(str.lower,  mydata.columns)  #  Convertit  tous  les
noms des variables de la table mydata en miniscules
mydata.columns  =  map(str.upper,  mydata.columns)  #  Convertit  tous  les
noms des variables de la table mydata en majuscules
```

4.7.4. Transformer une colonne en un index dans un dataframe

Soit la variable mydata précédemment définie. Transformer la variable firt_name en index.

```
mydata = mydata.set_index('first_name')
print(mydata.head())
```

L'indexation numérique est ainsi remplacée par l'indexation en caractère (bien que l'indexation numérique reste encore active). L'avantage de l'indexation par une variable est qu'il facilite la sélection ou la suppression des observations en indiquant l'index (voir section sélection ou supression des obervations)

4.7.5. Extraire les noms des colonnes sous forme de liste

Exemple : Soit la table df définie précédemment: extaire les noms des colonnes

```
col=df.columns.values # extraire les noms de toutes colonnes.
col=df.iloc[:,0:4].columns.values # extraire les noms des 4 premières colonnes.
```

Ces noms peuvent être utilisés dans d'autres contextes comme les assigner à une autre table contenant les mêmes données.

Penser également à utiliser: feature_names. Par exemple pour une base données python comme iris (qui sont généralement composées de plusieurs sous bases et les noms des features. On fait:

```
print(iris) pour voir le contenu
col=iris.feature_names.
```

4.7.6. Types des variables dans un dataframe

4.7.6.1. Déterminer le type des variables : la fonction dtypes

Le type d'une variable représente la nature de cette variable (numérique ou caractère). Pour connaitre le type d'une variable dans un dataframe, on utilise la fonction dtypes. Exemple :

```
mydata.dtypes # renvoie le type de chaque variable présente dans la table de données mydata.
mydata['myvar1'] # renvoie le type de la variable myvar1 de la table mydata
mydata['myvar1', 'myvar2'] # renvoie le type des deux variables myvar1 et myvar2
```

NB : Les variables en caractères sont représentées par le type « object », les variables numériques entières par « int », les variables numériques décimales par « float ».

4.7.6.2. Convertir le type d'une variable dans un dataframe : la fonction astype()

A l'image des objets ouvert de python, on peut convertir les variables dataframes d'un type à un autre. Pour cela, on utilise la fonction astype(). Exemple :

Soit la table de données suivante :

```
df = pandas.DataFrame(dict(myvar1 = pandas.Series(['1.0','1', '2.3','5', '3.1']), myvar2 = pandas.Series(['1.0','foo', '6', ' ' , '4.3'])))
```

La table df est constituée de deux variables et 5 observations.

```
df.dtypes # renvoie le type des variables
```

myvar1 est contituée de valeurs en chiffres se présentant sous format caractères; myvar2 est un mélange de valeurs en chiffres (se présentant sous forme de caractères) et de valeurs caractères pures (y compris l'espace vide).

Pour convertir myvar1 en type numérique, on fait comme suit :

```
df['myvar1']=df['myvar1'].astype(float) # Change le type de myvar1 de object (str) vers float.
```

Par contre la conversion de myvar2 en type numérique est un peu plus délicat. En effet, à cause de la présence des valeurs non numériques, en utilisant la fonction astype(), python renvoie une erreur. Dans tous les cas, la conversion échoue. C'est pourquoi, il faut utiliser d'autres fonctions pour la conversion. On peut par exemple utiliser la fonction « convert_objects(convert_numeric=True) ». Ainsi on a :

```
df['myvar2']=df['myvar2'].convert_objects(convert_numeric=True)        # convertit myvar2 en numérique.
```

NB : Avec la fonction convert_objects() toutes les valeurs qui ne sont pas sous forme de chiffres sont remplacées par une valeur manquante représentées par NaN.

Noter aussi qu'on pouvait utiliser la fonction convert_objects() pour convertir myvar1.

Attention : il semble que la fonction convert_objects soit dépréciée. Son utilisation n'est donc plus recommandée(voir le lien suivant http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.convert_objects.html

4.7.7. Création de variable

Soit la table suivante:

```
raw_data = {'regiment': ['Nighthawks', 'Nighthawks', 'Nighthawks', 'Nighthawks', 'Dragoons', 'Dragoons', 'Dragoons', 'Dragoons', 'Scouts', 'Scouts', 'Scouts', 'Scouts'],
        'company': ['1st', '1st', '2nd', '2nd', '1st', '1st', '2nd', '2nd','1st', '1st', '2nd', '2nd'],
        'name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze', 'Jacon', 'Ryaner', 'Sone', 'Sloan', 'Piger', 'Riani', 'Ali'],
        'preTestScore': [4, 24, 31, 2, 3, 4, 24, 31, 2, 3, 2, 3],
        'postTestScore': [25., 94, 57, 62, 70, 25, 94, 57, 62, 70, 62, 70]}
df = pandas.DataFrame(raw_data, columns = ['regiment', 'company', 'name', 'preTestScore', 'postTestScore'])
print(df)
```

4.7.7.1. Création des variables avec les opérateurs simples

Créons une nouvelle variable égale à la différence entre postTestScore et preTestScore. On a

```
df['diff_score'] = df['postTestScore']-df['preTestScore'] # différence entre deux variables
df['diff_scoreX2'] = 2*(df['postTestScore']-df['preTestScore']) ### Une variable égale au double de la différence
df['diff_score2'] = (df['postTestScore']-df['preTestScore'])**2 #### Une variable égale à la différence au carré
print(df)
```

4.7.7.2. Création des variables avec les fonctions mathématiques

Pour créer des variables avec des opérations plus avancées (log, exponentielle, cosinus,...), il faut utiliser la fonction math de numpy. Toutefois il faut noter que certaines fonctions de math ne marchent que sur des variables en float pas sur les entiers (ex: radian). Il faut donc veiller à convertir ces variables avant d'utiliser ces fonctions maths en utilisant la fonction float()

Exemple: Créons la variable log_duretot égale au log de duretot qui représentent le log et l'exponentiel de la variable durée totale duretot. On a

```
from numpy import math
mydata['log_duretot'] = numpy.log(mydata['duretot']) # Sans problème de conversion
mydata['exp_duretot'] = numpy.exp(mydata['duretot'])
mydata['exp_diff'] = numpy.exp(mydata['propsurfa']-mydata['proppro'])
```

Noter par exemple que mydata['log_duretot'] = math.log(mydata['duretot']) aurait renvoyé un message d'erreur à cause du problème de conversion.

Il faut noter qu'il existe plusieurs fonctions statistiques et mathématiques qui peuvent être utilisées lors de la création des variables. La liste ci-dessous illustre quelques fonctions :

Fonction statistique	**Description**
count()	Nombre d'éléments d'un objets (ou series, dataframe, liste, etc)
sum()	Somme des éléments
prod()	Produit des éléments
cumprod()	Produit cumulé des éléments
mean()	Moyenne des éléments
median()	Médiane des éléments

mad()	Deviation absolue moyenne
mode()	Calcule le mode
diff()	Différence entre les éléments successifs du vecteur
var()	variance des éléments du vecteur
std()	écart-type des éléments du vecteur
min()	minimum des éléments du vecteur
max()	maximum des éléments du vecteur
sem()	Erreur standard de la moyenne
skew()	skewness
kurt()	kurtosis
cummin()	minimum cumulatif des éléments
cummax	maximum cumulatif des éléments
quantile()	quantiles empiriques du vecteur(0% 25% 50% 75% 100%)
summary()	Statistiques descriptives

Les exempls ci-dessous illustrent quelques utilisations de ces fonctions:

Soit la table:

```
data = pd.DataFrame.from_csv('phone_data.cv')
data['item'].count() # compte le nombre d'observation de la variable item
data['duration'].max() # maximum de la variable duration
data['duration'][data['item'] == 'call'].sum() # Somme de la variable duration lorsque item est call
data['month'].value_counts() # Compte le nombre de répétitions pour chaque valeur de mois
data['network'].nunique() # Compte le nombre de valeurs qui sont dupliquées dans la table.
```

4.7.7.3. Création des variables par recodage : recodage conditionnel

Recodage conditionnel simple : utilisation de la fonction numpy.where()

Il s'agit d'un recodage d'une variable numérique en une variable caractère(ou numérique) avec une simple condition.

Exemple 1:

```
mydata['proximitech'] = numpy.where(mydata['proploin']>=0.50, 'loin', 'proche') # caractères
```

On crée une variable proximitech qui prend deux valeurs loin ou proche selon que la variable proploin a une valeur supérieure à 0.5 (recodage en variable caractère)

Exemple 2 :

```
mydata['proximitenum'] = numpy.where(mydata['proploin']>=0.50, 2, 1) # numérique
```

On crée une variable proximitenum qui prend deux valeurs 2 ou 1 selon que la variable proploin a une valeur supérieure à 0.55 (recodage en variable numérique)

Recodage avec plusieurs conditions

1- Utilisation de la condition elif

Exemple :

Soit la table suivante

```
dico = {'student_name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze', 'Jacon', 'Ryaner', 'Sone', 'Sloan', 'Piger', 'Riani', 'Ali'],
        'test_score': [76, 88, 84, 67, 53, 96, 64, 91, 77, 73, 52, numpy.NaN]}
mydata = pandas.DataFrame(dico, columns = ['student_name', 'test_score'])
```

Créons une nouvelle variable appelée grades contenant A, B, C, etc…

On a par exemple

```
grades = [] # liste vide(au départ)
for valeur in mydata['test_score']:
    if valeur > 95:
        grades.append('A')
    elif valeur > 90: # Cette condition veut dire implicitement valeur >90 and valeur <=95
        grades.append('A-')
    elif valeur > 85:
        grades.append('B')
    elif valeur > 80:
        grades.append('B-')
    elif valeur > 75:
```

```
        grades.append('C')
    elif valeur > 70:
        grades.append('C-')
    elif valeur > 65:
        grades.append('D')
    elif valeur > 60:
        grades.append('D-')
    else:
        grades.append('Failed')
# Ajout de la colonne grades à la table
mydata['grades'] = grades
print(mydata)
```

2- Utilisation de la fonction categorical

Soit la table suivante

```
df=pandas.read_csv("SAheart.csv",sep=';', header=0, encoding ='utf-8')
# ou encoding ='Latin-1'
```

Recodons la variable famhist en une variable catégorielle numérique. On fait :

```
df['famhist_ord']    =    pandas.Categorical(df['famhist']).labels    #
Transforme la variable famhist en variable catégorielle numérique
print(df)
```

3- Utilisation d'une fonction def avec la fonction apply()

Soit la table suivante

```
dico = {'patient': [1, 1, 1, 2, 2],
        'obs': [1, 2, 3, 1, 2],
        'treatment': [0, 1, 0, 1, 0],
        'score': ['strong', 'weak', 'normal', 'weak', 'strong']}
mydata   =   pandas.DataFrame(dico,   columns   =   ['patient',   'obs',
'treatment', 'score'])
```

On va recoder la variable score en une variable numérique nommée score_num en définisant une fonction. Pour cela, recodons la variable score telle que weak=1, normal=2 et strong=3. Ainsi, on a:

```
def mycode(x):
    if x=='strong':
        return 3
    if x=='normal':
        return 2
    if x=='weak':
        return 1
mydata['score_num'] = mydata['score'].apply(mycode)
```

La fonction apply() permet d'appliquer une fonction sur la table de donnée dataframe

```
print(mydata)
```

4.7.7.4. Recodage d'une variable continue (par intervalle de valeurs)

Soit la table suivante:

```
dico = {'regiment': ['Nighthawks', 'Nighthawks', 'Nighthawks', 'Nighthawks', 'Dragoons', 'Dragoons', 'Dragoons', 'Dragoons', 'Scouts', 'Scouts', 'Scouts', 'Scouts'],
        'company': ['1st', '1st', '2nd', '2nd', '1st', '1st', '2nd', '2nd','1st', '1st', '2nd', '2nd'],
        'name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze', 'Jacon', 'Ryaner', 'Sone', 'Sloan', 'Piger', 'Riani', 'Ali'],
        'preTestScore': [4, 24, 31, 2, 3, 4, 24, 31, 2, 3, 2, 3],
        'postTestScore': [25, 94, 57, 62, 70, 25, 94, 57, 62, 70, 62, 70]}
mydata = pandas.DataFrame(dico, columns = ['regiment', 'company', 'name', 'preTestScore', 'postTestScore'])
```

Recodons la variable postTestScore tel que 0 to 25, 25 to 50, 60 to 75, 75 to 100 et ces intervalles telque Low, Okay, Good et Great.

Alors on fait:

```
bornes = [0, 25, 50, 75, 100]
noms = ['Low', 'Okay', 'Good', 'Great']
mydata['categories'] = pandas.cut(mydata['postTestScore'], bornes, labels=noms)
print(mydata)
Compter chaque valeur de la variable catégorie
x=pandas.value_counts(mydata['categories'])
print(x)
```

4.7.8. Discrétisation d'une variable quantitative

Pour la discrétisation d'une variable quantitative, il est d'un bon usage de définir les bornes des classes à des quantiles, plutôt également espacées, afin de construire des classes d'effectifs sensiblement égaux. Ceci est obtenu par la fonction qcut(). La fonction qcut() propose par défaut des bornes équiréparties à moins de fournir une liste de ces bornes.

Exemple :

```
df["AgeQ"]=pandas.qcut(df.Age,3,labels=["Ag1","Ag2", "Ag3"])
df["PrixQ"]=pandas.qcut(df.Prix,3,labels=["Pr1","Pr2","Pr3"])
df["PrixQ"].describe()
```

4.7.9. Convertir une variable catégorielle en une variable binaire

Il existe plusieurs méthodes pour la conversion d'une variable qualitative en caractères en des variables binaires. La première méthode est l'utilisation de la fontion get_dummies() et la seconde méthode est l'utilisation de la fonction patsy.dmatrix()

4.7.9.1. Utilisation de la fonction get_dummies()

Exemple: soit les données suivantes

```
dico = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'last_name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze'],
        'sex': ['male', 'female', 'male', 'female', 'female']}
mydata= pandas.DataFrame(dico, columns = ['first_name', 'last_name',
'sex'])
```

Créons une variable binaire pour chaque modalité de la variable sex

```
muettes= pandas.get_dummies(mydata['sex'], prefix='sex_dummy')) # crée
d'abord une table constituée des dummies qu'il faut ensuite joindre à
mydata
mydata = pandas.concat([mydata, muettes], axis=1) # axis = 1 indique
que nous nous référons à une colonne, et non pas une ligne
# Méthode alternative pour joindre
mydata = mydata.join(muettes)
print(mydata)
```

4.7.9.2. Utilisation de la fonction patsy.dmatrix()

Exemple

Soit la table suivante:

```
dico = {'patient': [1, 1, 1, 0, 0],
        'obs': [1, 2, 3, 1, 2],
        'treatment': [0, 1, 0, 1, 0],
        'score': ['strong', 'weak', 'normal', 'weak', 'strong']}
mydata= pandas.DataFrame(dico, columns = ['patient', 'obs',
'treatment', 'score'])
```

Convertissons la variable score en variables binaires

```
x=patsy.dmatrix('score', mydata, return_type='dataframe') # cette
méthode considère la première modalité comme la constante (contenant
uniquement 1)
print(x)
# joindre à la table initiale
mydata = pandas.concat([mydata, x], axis=1)
print(mydata)
```

4.7.9.3. Utilisation de la fonction patsy.dmatrix() pour une variable numérique

La fonction patsy.dmatrix peut aussi être utilisée pour une variable numérique pour la transformer en variables binaires. Exemple :

Soit la table suivante :

```
dico = {'countrycode': [1, 2, 3, 2, 1]}
mydata = pandas.DataFrame(dico, columns = ['countrycode'])
Convertir la variable countrycode en trois variable binaires
x=patsy.dmatrix('C(countrycode)-1', mydata, return_type='dataframe')
# faire une jointure
mydata = pandas.concat([mydata, x], axis=1)
print(mydata)
```

4.7.10. Renommer les modalités d'une variable catégorielles

Le recodage des variables qualitatives ou renommage des modalités est obtenu simplement en utilisant cat.rename_categories(). Exemple :

```
df["Surv"]=df["Surv"].cat.rename_categories(["Vnon","Voui"])
df["Classe"]=df["Classe"].cat.rename_categories(["Cl1","Cl2","Cl3"])
df["Genre"]=df["Genre"].cat.rename_categories(["Gfem","Gmas"])
df["Port"]=df["Port"].cat.rename_categories(["Pc","Pq","Ps"])
```

4.7.11. Suppression de variables

La syntaxe générale de suppression d'une variable est:

```
myDataFrame.drop(labels, axis=1, level=None, inplace=True,
errors='raise')
```

Ci-dessous quelques exemples:

4.7.11.1. Supprimer une seule variable

```
df = df.drop('myvar', axis=1,inplace=True) # suppression de la colonne
myvar du tableau.
```

Nb : axis=1 signifie qu'on s'intéresse aux colonnes et non aux lignes

Penser aussi à utiliser l'option inplace=False lorsque l'on veut stocker la nouvelle table sous un autre nom. ex: df1 = df.drop('myvar', axis=1,inplace=False)

4.7.11.2. Suppression de plusieurs variables

```
df1 = df.drop(['myvar1', 'myvar2', 'myvar3'], axis=1) # # penser aussi
à utiliser l'option inplace=True au besoin
```

On peut aussi supprimer les variables en indiquant leur indice.

```
df1 =df.drop(df.columns[[0, 1, 3]], axis=1)   # Suppression des variables correspondant aux indices 0, 1 et 3
```

4.7.11.3. Supprimer toutes les variables dont le nom commence par un mot

Soit la table

```
dico = {'regiment': ['Nighthawks', 'Nighthawks', 'Nighthawks', 'Nighthawks', 'Dragoons', 'Dragoons', 'Dragoons', 'Dragoons', 'Scouts', 'Scouts', 'Scouts', 'Scouts'],
        'company': ['1st', '1st', '2nd', '2nd', '1st', '1st', '2nd', '2nd','1st', '1st', '2nd', '2nd'],
        'name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze', 'Jacon', 'Ryaner', 'Sone', 'Sloan', 'Piger', 'Riani', 'Ali'],
        'preTestScore': [4, 24, 31, 2, 3, 4, 24, 31, 2, 3, 2, 3],
        'postTestScore': [25, 94, 57, 62, 70, 25, 94, 57, 62, 70, 62, 70]}
mydata= pandas.DataFrame(dico, columns = ['regiment', 'company', 'name', 'preTestScore', 'postTestScore'])
```

Suppression toutes les variables dont les noms commencent par 'pre'

Astuce: on sélectionne les variables dont les noms de ne commencent pas par pre:

```
mesvar = [nomvar for nomvar in mydata.columns if nomvar.lower()[:3] != 'pre'] # les 3 premieres lettres (d'indice 0, 1 et 2) de nomvar différentes de pre (astucieux)
# suppression par sélection
mydata=mydata[mesvar]
print(mydata) # la variable preTestScore n'est pas gardée (donc elle est supprimée)
```

4.7.12. Récupérer les valeurs d'une variable et les stocker sous forme de liste

Soit la table mydata précédement définie.

Récupérons les valeurs de preTestScore sous forme de liste:

```
z= list(mydata['postTestScore'])
```

Si on voulait récupérer les valeurs sans duplications, on suit l'une des deux méthodes suivantes:

Première méthode

```
z1= list(mydata['postTestScore'].unique())
print(z1)
```

Deuxième méthode

z2=list(set(mydata['postTestScore']))

print(z2)

4.7.13. Convertir une date en chaine de caractères en une date reconnue

Soit la table de donnée suivante:

```
mydata = pandas.DataFrame.from_csv('phone_data.csv')

print(mydata.head(n=5))
```

On voit que la date n'est pas sous format standard mais plutôt sous format texte qu'il va falloir convertir.

On va d'abord convertir la date en format numérique en utilisant le module dateutil

```
import dateutil
mydata['date'] = mydata['date'].apply(dateutil.parser.parse, dayfirst=True) # converti la date en format numérique (en inversant l'ordre jour mois et année)
print(mydata.head(n=5))
```

4.8. Opérations sur les observations

4.8.1. Sélectionner des observations

4.8.1.1. Sélection des observations à partir de leur indice

Soit la table de données nommée mydata. On peut faire les sélections suivantes :

```
mydata.iloc[1:2] # Sélectionner de la ligne 1 à la ligne 2
mydata.iloc[:2] #  Sélectionner toutes les lignes jusqu'à 2 (exclue) ; Attention à l'indie 0
mydata.iloc[2:] # Sélectionner à partir de la ligne 2 (incluse)
mydata.ix['Maricopa'] # Sélectionner une ligne selon l'intitulé de l'index (ex:Maricopa)

mydata.loc[:'Maricopa'] # Sélectionner toutes les observations dont l'intitulé de l'index est donné (ex:Maricopa)
mydata.ix['Yuma', 'coverage'] # Sélectionner une valeur située dans une cellule (c'est à dire la valeur d'une variable pour un individu donné). Ex: la valeur de coverage pour Yuma
```

4.8.1.2. Sélction des observations selons les valeurs d'une ou de plusieurs variables (sélection conditionnelle)

Exemples :

Sélectionner toutes les observations pour lesquels coverage est supérieure à 50

```
mydata[mydata['coverage'] > 50]
```

Sélectionner toutes les observation pour lesquel coverage est supérieur à 90 ou coverage est inféreir à 50

```
mydata[(mydata['coverage'] > 90) | (mydata['coverage'] < 50)]
```

Sélectionner toutes les observations pour lesquel coverage est supérieur à 50 et coverage est inféreir à 90

```
mydata[(mydata['coverage'] < 90) & (mydata['coverage'] > 50)]
```

Sélectionner les observations pour lesquelles il n'y a pas de valeurs manquantes. Ex: sélectionner les observations pour lesquelle coverage et reports ne sont pas manquantes

```
mydata[mydata['coverage'].notnull() & mydata['reports'].notnull()]
```

Sélectionner toutes les observations pour lesquelle name est différent de 'Jason'

```
mydata[~(mydata['name'] == 'Jasons')]
```

4.8.1.3. Sélectionner des observations par tirage aléatoire

On peut aussi effectuer des échantillonnages sur la table de données en faisant des tirages aléatoires.

Mais avant de faire le tirage, il faut fixer le seed pour pouvoir retrouver les mêmes obervations tirées (et retrouver les mêmes résultats).

Exemple :

```
numpy.random.seed(seed=2016) # fixe le seed à 2016
numpy.random.seed(seed=None) # annule le seed
```

Mise en oeuvre du tirage

Exemple: soit la table

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'last_name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze'],
        'age': [42, 52, 36, 24, 73],
        'preTestScore': [4, 24, 31, 2, 3],
        'postTestScore': [25, 94, 57, 62, 70]}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'last_name',
'age', 'preTestScore', 'postTestScore'])
print(df)
```

Tirage sans remise avec un nombre fixe n d'observations

```
df1=df.sample(n=50) # Tirage aléatoire sans remise de 50 obervations
print(df1)
df2=df.sample(frac=0.5) # Tirage sans remise avec une proportion (
Ex : 50% obervations)
print(df2)
```

Tirage avec remise avec un nombre fixe n d'observations

```
df3=df.sample(n=50, replace=True) # Tirage avec remise de 50
obervations
print(df3)
df4=df.sample(frac=0.5, replace=True) # Tirage avec remise avec une
proportion (Ex : 50% obervations)
print(df4)
```

Remarque : On pouvait aussi utiliser numpy pour faire les tirage aléatoires.

Exemple : tirage d'un échantillon de m observations parmi n

```
d1 = numpy.random.choice(df['preTestScore'],size=5,replace=False)
#sans remise
print(d1)
d2 = numpy.random.choice(df['preTestScore'],size=5,replace=True) #avec
remise
print(d2)
```

Malheureusement avec ce tirage avec cette fonction une seule variable doit être indiquée (pas toute la table).

4.8.2. Trier les observations

Pour trier les valeurs des variables dans un data frame, on utilise la fonction sort_values(). Ci-dessous quelques exemples :

Soit la table suivante:

```
data = {'name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [1, 2, 1, 2, 3],
        'coverage': [2, 2, 3, 3, 3]}
df = pandas.DataFrame(data, index = ['Cochice', 'Pima', 'Santa Cruz',
'Maricopa', 'Yuma'])
```

Tri sur une seule variable :

```
df=df.sort_values(by=['coverage'],  ascending=[True])  #  Trier  les observations par valeur croissante de coverage
df=df.sort_values(by=['coverage'],  ascending=[False])  #  Trier  les observations par valeurs décroissantes de coverage
```

Tri sur plusieurs variables :

```
df=df.sort_values(by=['coverage',    'reports'],    ascending=[False, False]) # Tri par ordre décroissant sur les deux variables
df=df.sort_values(by=['coverage', 'reports'], ascending=[True, False])
# Tri par ordre croissant sur coverage et décroissant sur reports
df=df.sort_values(by=['coverage', 'reports'], ascending=[True, True])
# Tri par ordre croissant sur les deux variables
df=df.sort_values(by=['coverage', 'reports'], ascending=[False, True])
# Tri par ordre décroissant sur coverage et croissant sur reports
```

4.8.3. Traitement des observations avec valeurs manquantes

4.8.3.1. Création d'une table avec des valeurs manquantes

Exemple :

```
raw_data = {'first_name': ['Jason', numpy.nan, 'Tina', 'Jake', 'Amy'],
        'last_name': ['Miller', numpy.nan, 'Ali', 'Milner', 'Cooze'],
        'age': [42, numpy.nan, 36, 24, 73],
        'sex': ['m', numpy.nan, 'f', 'm', 'f'],
        'preTestScore': [4, numpy.nan, numpy.nan, 2, 3],
        'postTestScore': [25, numpy.nan, numpy.nan, 62, 70]}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'last_name', 'age', 'sex', 'preTestScore', 'postTestScore'])
print(df)
```

4.8.3.2. Supprimer toutes les observations avec valeurs manquantes

Supprimer toutes les observations contenant une valeur manquante (au moins sur une variable)

Exemple :

```
df.dropna(how="all", inplace=True) # lorsqu'on veut garder la même table
df2 = df.dropna(how="all",, inplace=False) # si l'on veut créer une nouvelle table
print(df2)
```

Supprimer une ligne contenant uniquement des valeurs manquantes

Exemple :

```
df= df.dropna(how='all')
```

Supprimer une colonne contenant uniquement des valeurs manquantes

```
df=df.dropna(axis=1, how='all')
```

Remplacer toutes les valeurs manquantes d'une table par 0

```
df=df.fillna(0)
```

Remplacer une valeur spécifique par une valeur manquante dans toute la table

```
df=df.replace(-999, numpy.nan) # Remplace les -999 par une valeur manquante
```

On peut aussi appliquer cette fonction à d'autres valeur pas uniquement les valeurs manquantes

```
df=df.replace(42, 1000)
```

```
print(df)
```

Remplacer la valeur manquante par la moyenne

Exemple : la variable preTestScore

```
df=df["preTestScore"].fillna(df["preTestScore"].mean(), inplace=True)
```

Remplacer la valeur manquante par la médiane

df=df.fillna(df.median())

Par la modalité "médiane" de AgeQ

df.AgeQ=df["AgeQ"].fillna("Ag2")

Remplacement par le port le plus fréquent

```
df["Port"].value_counts()
df.Port=df["Port"].fillna("Ps")
```

4.8.4. Identifier l'observation qui a la valeur maximale sur une variable

Exemple: Identifier l'observation qui a le maximum de preTestScore de la table mydata (en renvoyant son index)

```
x=mydata['preTestScore'].idxmax()
print(x)
```

4.8.5. Suppression d'observations

4.8.5.1. Suppression des observations sur la base des indices

La manière la plus simple de supprimer une observation est d'invoquer son indice.

Exemple:

```
dico = {'name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [4, 24, 31, 2, 3]}
mydata  =  pandas.DataFrame(dico,  index  =  ['Cochice',  'Pima',  'Santa
Cruz', 'Maricopa', 'Yuma'])
```

Supprimons les observations dont l'index est 'Cochice', 'Pima'

mydata.drop(['Cochice', 'Pima']) # supprime les observations indicées Cochine et Pima.

NB : La suppression peut aussi se faire en se basant sur les indices numériques. Dans ce cas, on remplace l'intitulé des indices par les valeurs des indices sous forme de liste.

4.8.5.2. Suppression des observations selon une condition

Suppression par sélection :

Exemple : Sélection des observations des individus dont le nom est différent de 'Tina' (donc suppression des observations correspondant à tina)

```
mydata = mydata[mydata.name != 'Tina']
```

4.8.6. Identification et Suppressions des obervations dupliquées dans une table

4.8.6.1. Identification des observations dupliquées

```
x=mydata.duplicated()  #  recherche  la  duplication  sur  l'ensemble  des
variables de la table
print(x) # les obervations dupliquées prennent True
x=mydata['reports'].duplicated()  #  recherche  la  duplication  sur  la
variable reports
print(x) # les obervations dupliquées prennent True
y=mydata[['reports',   'coverage']].duplicated()   #   Recherche   la
duplication sur deux variables reports et coverage
```

4.8.6.2. Suppression des observations dupliquées

```
mydata.drop_duplicates() # supprime les duplication sur toute la table
z=df['reports'].drop_duplicates()  #  supprime  la  duplication  sur  la
variable reports
z=df[['reports',   'coverage']].drop_duplicates()   #   suprime   la
duplication sur deux variables.
```

NB : Pour la suppression de la duplication sur une variable, on peut aussi utiliser la fonction unique() ou le fonction set(). Exemples :

```
z1= list(mydata['reports'].unique())
z2=list(set(mydata['reports']))
```

4.8.7. Comptage des valeurs dans une table

Soit la table de données suivante qui donne le nombre de morts par régiment d'armés et par année. Les régiments ici sont les variables.

```
x=pandas.read_csv("Ndeaths_data.csv",sep=';', header=0)
print(x)
```

4.8.7.1. Comptage des valeurs sur une seule variable

Exemple : Comptons le nombre fois où chaque valeur se repète pour la variable guardCorps

```
y=pandas.value_counts(x['guardCorps'].values, sort=False)
print(y)
```

4.8.7.2. Comptage sur plusieurs variables

Exemple : Comptons le nombre de fois où chaque nombre de morts se repète par régiment c'est-à-dire pour chaque variable de table (à l'exception de la variable year).

Enlevons d'abord la variable year(qui n'est pas concerné par le calcul)

```
tab=x.drop('year', axis=1)
n1tab = tab.apply(pandas.value_counts).fillna(0) # compte les valeurs
pour chaque variable.
print(n1tab)
```

4.8.8. Créer un rang pour les obervations

Soit la table suivante:

```
dico = {'name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [4, 24, 31, 2, 3],
        'coverage': [25, 94, 57, 62, 70]}
mydata = pandas.DataFrame(dico, index = ['Cochice', 'Pima', 'Santa
Cruz', 'Maricopa', 'Yuma'])
```

Créons une nouvelle variable nommée rang en triant par ordre croissant la variable coverage.

```
mydata['rang'] = mydata['coverage'].rank(ascending=1) # du +petit au
+grand
```

```
mydata['rang'] = mydata['coverage'].rank(ascending=0) # du +grand aux + petit
print(mydata)
```

4.9. Construire une boucle DO LOOP (sur les variables ou sur les observations)

Exemple: soit la table suivante:

```
dico = {'regiment': ['Nighthawks', 'Nighthawks', 'Nighthawks', 'Nighthawks', 'Dragoons', 'Dragoons', 'Dragoons', 'Dragoons', 'Scouts', 'Scouts', 'Scouts', 'Scouts'],
        'company': ['1st', '1st', '2nd', '2nd', '1st', '1st', '2nd', '2nd','1st', '1st', '2nd', '2nd'],
        'name': ['Miller', 'Jacobson', 'Ali', 'Milner', 'Cooze', 'Jacon', 'Ryaner', 'Sone', 'Sloan', 'Piger', 'Riani', 'Ali'],
        'preTestScore': [4, 24, 31, 2, 3, 4, 24, 31, 2, 3, 2, 3],
        'postTestScore': [25, 94, 57, 62, 70, 25, 94, 57, 62, 70, 62, 70]}
df = pandas.DataFrame(dico, columns = ['regiment', 'company', 'name', 'preTestScore', 'postTestScore'])
```

On veut faire une boucle do loop sur les variables ou les observations de cette table. On peut utiliser la structure de code suivante (qui doit être adaptée au cas étudié)

```
for names, values in df.iteritems(): # récupère le nom et les valeurs de chaque variable pour la première ligne ( revoir la fonction iteritems())
    print ('{name}: {val}'.format(name=names, val=values[0]))
```

Selon les cas, on peut soit récupérer le nom de la variable ou récupérer la valeur (selon les besoins)

Par exemple, on veut multiplier toutes les variables par 10. On fait:

```
for names, values in df.iteritems()
    df['{name}'.format(name=names)]
=10*df['{name}'.format(name=names)]
```

Ne pas oublier l'indentation. On peut effectuer la même opération avec la valeur values. Toutefois, pour faire une boucle explicite pour les observations, on doit utiliser la boucle iterrows.

4.10. Calculer la somme par colonne ou par ligne et calcul du cumul

Soit la table:

```
df = pandas.DataFrame({'a': [1,2,3], 'b': [2,3,4], 'c':['dd','ee','ff'], 'd':[5,9,1]})
```

4.10.1. Calcul de la somme par ligne

Pour faire la somme par ligne d'une table, on utilise simplement la fonction sum() avec l'option axis=1.

```
df['e'] = df.sum(axis=1) # calcul la somme par ligne (sur toutes la tables) et crée la variable e.
print(df)
```

Pour la somme sur quelques variables, on supprime les variables non concernées. Ex :

```
col_list= list(df) # liste de toutes variables de la table df
col_list.remove('d') # Exclusion de la colonne
df['e'] = df[col_list].sum(axis=1) # calculi de la somme sur les variable considérées.
print(df)
```

4.10.2. Calcul de la somme par colonne

Pour faire la somme d'une variable (colonne), on utilise simplement la fonction sum() avec l'option axis=0.

Exemple:

```
df['f'] = df['e'].sum(axis=0) # total  de la variable e sur tout l'échantillon et crée un variable f.
```

4.10.3. Calcul de la somme cumulée par ligne ou par colonne

Pour calculer la somme cumulée d'une variable, on utilise la fonction cusum() avec axis=0. Pour la ligne, on fait axis=1

```
df['g'] = df['f'].cumsum(axis=0) # somme cumulée de f
```

Il faut savoir qu'on peut calculer l'ensemble de ces statistiques par sous-groups en utilisant la fonction groupby (voir ci-dessous)

4.11. Calcul de valeurs par groupe et agrégation des données dans une table

Pour faire l'aggréation des valeurs d'une variable on se sert de la fonction groupby() ou de la fonction agg()

4.11.1. Utilisation de la fonction groupby()

La fonction groupby permet de faire l'aggrégation d'une variable y selon les modalités d'une variable catégorielle. Plusieurs méthodes de groupby existent: mean(), max(), count() ... (presque toutes les fonctions statistiques usuelles).

Exemple: soit la table suivante:

```
mydata=pandas.read_excel("phone_data.xlsx",sheetname="data",  header=0,
parse_cols="A:G")
```

Présentons quelques utilisations de la fonction groupby:

```
mydata.groupby('month').first() # renvoie la première observation de
la variable month
mydata.groupby('month')['duration'].sum() # Fait la somme de duration
par month
mydata.groupby('month')['date'].count() # compte le nombre de dates
pour chaque mois
mydata[mydata['item']  ==  'call'].groupby('network')['duration'].sum()
# Calcul la somme de duraction pour chaque valeur de network quand
l'item est égal à call
La fonction groupby() peut aussi etre utilisée pour plusieurs
variables de regroupement. Celle-ci doivent être spécifiée comme une
liste Exemple:
mydata.groupby(['month',  'item'])['date'].count()  #  le  nombre  de
comptage par date pour chaque item dans le mois
mydata.groupby(['month', 'network_type'])['date'].count() # le nombre
de comptage par date pour chaque netwrok_type dans le mois
```

4.11.2. Utilisation de la fonction groupby() combinée avec la fonction agg()

La fonction groupby telle que présentée ne permet de fournir qu'une seule statistique: sum(), mean() mais pas plusieurs à la fois. c'est pourquoi, il faut lui associer la fonction agg() lorsque l'on veut présenter plusieurs statistiques (sur une seule variable ou pour plusieurs variables)

Les lignes ci-dessous illsutrent les différentes manières d'utiliser agg()

Statistique pour pour plusieurs variable (avec une seule fonction par variable)

Ici, on a deux formulations possibles:

Première formulation:la valeur agrégée prend le même nom que la variable intiale

Exemple :

```
data.groupby(['month', 'item']).agg({'duration':sum,        # la somme
de duration pour chaque groupe
                                     'network_type':  "count",  #  le
nombre de network_type par groupe
```

```
                                     'date':  'first'})        #  la
première date par groupe
```

Deuxième formulation:la valeur agrégée ne prend pas le même nom que la variable intiale

Exemple :

```
mydata.groupby(['month',     'item']).agg({'duration':{'duration_sum':
'sum'},     # la somme de duration pour chaque groupe
                                     'network_type':
{'network_type_count':  'count'},  #  le  nombre  de  network_type  par
groupe
                                     'date':{  'date_first':  'first'}})
# la première date par groupe
```

Attention, dans l'optique d'un merge avec la table initiale, la seconde formulation n'est pas adaptée alors que pour la première formulation il faut renommer la valeurs agregée avant de faire le merge (voir plus bas pour les merges des valeurs agrégées).

4.11.3 Utilisation d'une formule dans la définition d'une fonction d'aggréation

Il arrive souvent que la statistique qu'on veut calculer ne soit pas direcement disponible en prédéfinie. Il faut alors spécifier une formule en utilisant la fonction lambda.

On définit d'abord une liste des fonctions d'agréagtion que l'on veut utiliser. Exemple:

```
agliste = {
    'duration':'sum', # somme de la duration
    'date': lambda x: max(x) # calcul la valeur maximum de la variable
date
}
```

On applique ensuite cette fonction à la table:

mydata.groupby('month').agg(agliste) # Crée alors la somme de duration et le maximum de la date.

4.11.4. Définition d'une agréagtion plus complexe: statistique pour pour plusieurs variable (avec plusieurs fonctions par variable)

Dans cette liste, on veut définir plusieurs fonctions pour plusieurs variables (sans que les fonctions ne soient les mêmes pour toutes les variables)

```
agliste = {
    'duration': { # Sélectionne la colone "duration"
        'total_duration':  'sum',   #  calcule  la  somme  et  nomme  ce
resultat 'total_duration'
```

```
        'average_duration': 'mean', # calcule la moyenne et nomme ce resultat 'average_duration'
        'num_calls': 'count' # calcule le nombre de valeurs et nomme ce num_cals
    },
    'date': {    # Sélectionne la colone date
        'max_date': 'max',   # calculi le maximum et nomme "max_date"
        'min_date': 'min', # Calcule le minimum
        'num_days': lambda x: max(x) - min(x)  # Calcule l'étendue de date par groupe ( utilisation d'une formule: function lamda).
    },
    'network': ["count", "max"]    # Calcule simultaneament deux résultats pour network (nombre et max)
}
#Exécution de l'agrégation
mydata[mydata['item'] == 'call'].groupby('month').agg(agliste) # Effectue l'agrégation sur les groups dont l'item est gal à call
```

4.11.5. Merging des valeurs agrégées à la table initiale

Il faut remarquer que chaque application de la fonction groupby crée une nouvelle table de donnée. L'assoctaion de cette table à la table initiale n'est pas directe. Car par défaut la fonction groupby transforme les variables de groupement en des indices de données. Ce qui fait qu'en faisant merge. On reçoit un message.

Ainsi, pour pouvoir faire merge, il faut spécifier l'option supplémentaire as_index=False lors de la définition de la fonction d'agréagtion.

Par exemple, calculons la moyenne de la variable duration par mois et par type de services (month-item.) et ajoutons ces informations à la table initiale. On suit les étapes suivantes:

```
gdata=mydata.groupby(['month', 'item'],as_index=False).agg({'duration':'mean'})  # calcule la moyenne de la variable duration
```

Comme dans la formulation ci-dessus la valeur agrégée créée prend le même nom que la variable initiale, il faut renommer cette variable avant de faire merging.

```
gdata.rename(columns={'duration': 'duration_mean'}, inplace=True) #renommer duration en duration_mean
df_merge1=pandas.merge(mydata, gdata, on=['month', 'item'], how='inner') # merging
```

NB: Pour faire le merging de plusieurs valeurs agrégées d'une seule variable, il vaut mieux ne pas utiliser la spécification par liste de ces valeurs. Sinon la table qui sera créée ne peut pas être mergée directement avec la table initiale car les positions des noms des colonnes ne sont ne coincident pas. Par conséquent le merging échoue. Pour plus de précaution, il vaut mieux utiliser l'agrégation et le merging une à une.

4.12. Opération sur tables de données

4.12.1. Fusion de tables

On distingue deux types de fusion de tables: fusion de tables par observation et fusion de table par variables.

La fusion de table par variables consiste à fusionner deux tables de données ayant les mêmes variables mais des observations différentes. Ce type de fusion est également appelé fusion verticale (appending) qui consiste à empiler les données de deux tables.

La fusion de table par observations consiste à fusionner deux tables de données ayant les mêmes observations mais de variables différentes. Ce type de fusion est également appelé fusion horizontale (merging) qui consiste à joindre les données de deux tables.

Il existe cependant plusieurs variantes qui combinent ces deux types de fusions de tables. Cette section a pour but de présenter les différentes méthodes de fusion sous python.

Soient les trois tables de données suivantes

```
#data1
raw_data = {
        'subject_id': ['1', '2', '3', '4', '5'],
        'first_name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
        'last_name':    ['Anderson',    'Ackerman',    'Ali',    'Aoni',
'Atiches']}
df_a    =    pandas.DataFrame(raw_data,    columns    =    ['subject_id',
'first_name', 'last_name'])
#Data2
raw_data = {
        'subject_id': ['4', '5', '6', '7', '8'],
        'first_name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
        'last_name':    ['Bonder',    'Black',    'Balwner',    'Brice',
'Btisan']}
df_b    =    pandas.DataFrame(raw_data,    columns    =    ['subject_id',
'first_name', 'last_name'])
# Data3
raw_data = {
        'subject_id':  ['1',  '2',  '3',  '4',  '5',  '7',  '8',  '9',  '10',
'11'],
        'test_id': [51, 15, 15, 61, 16, 14, 15, 1, 61, 16]}
df_n = pandas.DataFrame(raw_data, columns = ['subject_id','test_id'])
```

La première table de données contient les noms et prénoms de 5 individus dont les indentifiants vont de 1 à 5.

La deuxième table de données contient les noms et prénoms de 5 individus dont les indentifiants vont de 4 à 8.

La troisième table de données contient les informations sur la variable test-id de 10 individus dont les indentifiants vont de 1 à 11 (excluant 6).

Nous allons appliquer les différentes méthodes de fusions en considérant les trois tables

4.12.1.1. Fusion verticale ou juxtaposition de table (append)

```
D'une manière générale, on utilise la juxtaposition de table lorsque
les deux tables ont les mêmes variables mais les observations
différentes
# application
df_new = pandas.concat([df_a, df_b])
print(df_new)
```

L'append est fait sans prendre en compte les clés d'identification des individus.On remarque par exemple que subject_id 5 se retrouve dans les deux tables mêmes si ce n'est pas le même individu.

4.12.1.2. Fusion de deux tables sur les variables (sans clé d'identification)

```
D'une manière générale, la fusion de tables sur les variables est
utilisée lorsque les tables ont les mêmes observations mais les
variables différentes.
# aplication
df_new=pandas.concat([df_a, df_b], axis=1) # axis=1 signifie qu'on
s'interesse aux colonnes plutôt qu'aux lignes
```

Lorsqu'on utilise concat avec axis=1 on distingue plusieurs situations:

Lorsque les variables dans les deux tables sont les mêmes, les variables sont dupliquées dans la tables finale.

Lorsque les individus dans les deux tables ne sont pas les mêmes, dans la table finale est créée une correspondance ligne par ligne.

Cette fusion de table n'est donc pas adaptée dans beaucoup de situation. Il faut donc prendre beaucoup de précaution avant de l'utiliser.

Par exemple, il faut l'utiliser lorsqu'il existe une correspondance exacte entre les lignes dans les deux tables

4.12.1.3. Fusion de deux tables sur les variables avec clé d'identification (merge)

Contrairement à la fusion précédente qui n'utilise pas de clé, on peut réaliser réaliser une fusion de tables sur les variables en ajoutant une clé en utilisant merge. Par exemple considérons la table df_new définie telle que

```
df_new = pandas.concat([df_a, df_b])
```

Réalisons un merge entre df_new et la table def_n décrite en haut. Le merge s'effectue sur une clé d'indentification subject_id en utilisant l'option on= Cependant plusieurs situation peuvent aussi se présenter lorsqu'on effectue un merge sur deux tables.

En effet, il y a des observations qui sont présentes uniquement dans une des deux tables et des obervations présente dans les deux tables.

Face à cette situation, il faut donc ajouter des options supplémentaires pour sélectionner quelles observations doivet figurer dans la table finale.

Merger en gardant les individus se trouvant dans les deux tables

On a trois méthodes

```
#  Première méthode
df_merge1=pandas.merge(df_new, df_n, on='subject_id', how='inner') # Pour merger sur plus d'une variable on fait une liste on=['subject_id1','subject_id2']
# Deuxième méthode
df_merge2= pandas.merge(df_new, df_n, left_on='subject_id', right_on='subject_id')
# Troisième méthode
df_merge3=pandas.merge(df_new, df_n, on='subject_id') # à confirmer. Il semblerait que inner est l'option par défaut.
```

Merger en gardant toutes les observations (celles se trouvant dans les deux tables et celles venant uniquement d'une table)

```
df_merge_outer=pandas.merge(df_new, df_n, on='subject_id', how='outer')
print(df_merge_outer) # les observations ne se trouvant pas dans l'autre table auront des valeurs manquants pour ces variabls
```

Merger en gardant les obervation provenant de la table1 (y compris bien celle se trouvant à la fois dans les deux tables)

```
df_merge_innerl=pandas.merge(df_new, df_n, on='subject_id', how='left')
```

Merger en gardant les obervation provenant de la table2 (y compris bien celle se trouvant à la fois dans les deux tables)

```
df_merge_innerr=pandas.merge(df_new, df_n, on='subject_id', how='right')
```

Merger les observations en utilisant l'index comme clé

```
df_mergeind=pandas.merge(df_new, df_n, right_index=True, left_index=True)
```

Ce type de merging fait une correspondance ligne par ligne.

Par ailleurs les variables dupliquées seront suffixées par x dans la table 1 et y dans la table 2 contrairement à concat(, axis=1) qui ne met aucun suffixe.

Aussi, on peut modifier ces suffixes par défaut en ajoutant les options suffixes)().

Exemple:

```
datasuff= pandas.merge(df_a, df_b, on='subject_id', how='left', suffixes=('_left', '_right'))
# ou encore
df_mergeind=pandas.merge(df_new, df_n, right_index=True, left_index=True, suffixes=('_left', '_right'))
```

4.12.2. Reformattage de tables (reshape)

Exemple:

Soit la table suivante:

```
raw_data = {'patient': [1, 1, 1, 2, 2],
        'obs': [1, 2, 3, 1, 2],
        'treatment': [0, 1, 0, 1, 0],
        'score': [6252, 24243, 2345, 2342, 23525]}
df_long = pandas.DataFrame(raw_data, columns = ['patient', 'obs', 'treatment', 'score'])
#conversion en format wide
df_wide=df_long.pivot(index='patient', columns='obs', values='score')
print(df_wide)
```

4.12.3. Mettre une table sous format verticale

Par défaut les tables dataframe se présentent sous la forme horizontale. Mais certains modules d'analyse de données tels que scikit-sklearn exigent que les données soient sous forme vertciale. Ici, nous présentons quelques méthodes de transformation des données.

On a principalement deux méthodes pour mettre une table sous format verticlae : la fonction numpy.matrix() et la fonction .values

Exemple ; soit la table de données suivante

```
dico={'region': ['North', 'Yorkshire', 'Northeast', 'East Midlands', 'West Midlands','East Anglia','Southeast','Southwest','Wales','Scotland','Northern Ireland'],
      'alcohol': [6.47, 6.13, 6.19, 4.89, 5.63,4.52,5.89,4.79,5.27,6.08,4.02],
      'tobacco': [4.03, 3.76, 3.77, 3.34, 3.47,2.92,3.20,2.71,3.53,4.51, 4.56]
      }
df=pandas.DataFrame(dico, columns = ['region', 'alcohol', 'tobacco'])
print(df)
```

On veut mettre cette table sous format verticale.

Transformer toute la table

Première méthode: la fonction numpy.matrix

```
vdata1 = numpy.matrix(df)
print(vdata1)
Deuxième méthode: la méthode. values
vdata2 = df.ix[:,:].values
print(vdata2)
```

Transformer quelques variables de la table

```
vdata1 = numpy.matrix(df['alcohol']) # première méthode
vdata2 = df.ix[:,['alcohol']].values # deuxième méthode
#vdata2 = df.ix[:,'alcohol'].values # deuxième méthode (bis) examiner les résultats avec la première methode.
print(vdata1)
print(vdata2)
```

4.13. Standardiser les variables d'une table

Standardiser une variable c'est soustraire la moyenne et diviser par l'écart type.

Exemple :

Soit la table de donnée suivante:

```
df=pandas.read_excel("iris.xlsx",sheetname="data", header=0, parse_cols="A:F")
print(df.head(n=5))
print(df.tail(n=5))
```

Les quatres premières colonnes sont des variables quantitatives. On va donc extraire ces variables et on convertit les variables en format verticale car le module sklearn nécessite dans certains cas que les données soient transformées:

```
x = df.ix[:,0:4].values # on pouvait aussi utiliser numpy.matrix()
print(x)
# Ensuite on standaridze les variables comme suit:
from sklearn.preprocessing import StandardScaler
x = StandardScaler().fit_transform(x)
```

Chapitre 5 : Les analyses statistiques classiques sous Python

Ce chapitre présente la mise en application des analyses statistiques classiques sous Python. Nous présentons notamment les analyses statistiques univariées, les statistiques descriptives bivariées, les analyses d'association entre variables, les tests de comparaison de moyennes, les régressions linéaires et les régressions logistiques.

5.1. Paramétrage de python et chargement des modules usuels

Avant d'entamer les analyses, il est commode de charger en amont tous les modules potentiellement nécessaires mais aussi paramétrer les fonctionnalités générales de python. Ci-dessous les différents paramétrages :

```
import os, sys, pandas, numpy ,patsy , re, random ,scipy,statsmodels
random.seed(2016) # Fixation du seed
# -*- coding: utf-8 -*- # fixation de l'encodage
pandas.set_option('expand_frame_repr', False) # augmente le nombre de variable par page en faisant print(myData)
pandas.set_option('display.max_columns', 1000) # Fixe  le nombre de colonnes à aficher à 1000
pandas.set_option('display.max_row', 1000000) # Augmente le nombre de lignes à afficher
os.chdir("D: /R applications/data" ) # fixation répertoire de travail
```

5.2. Statistiques descriptives

Les calculs statistiques dans python se font généralement avec le module scipy ou le module statsmodels. Dans ce document, nous allons réaliser les statistiques descriptives ainsi que quelques analyses statistiques ordinaires en utilisant les deux modules (scipy et statsmodels) pour comparer leur convivialité.

5.2.1. Statistiques desriptives sur les variables quantitatives

Pour réaliser une statistiques descriptive sur les variables quantitatives, on peut utiliser la méthode describe de scipy.stats ou le module statsmodels.

5.2.1.1. Utilisation du scipy.stats

Exemple: Soit la table de donnée suivante:

```
mydata=pandas.read_excel("QUESTIONNAIRE_ACCEUIL.xlsx",sheetname="Feuil1", header=0, parse_cols="A:R")
print(mydata.head(n=5))
```

Calculons les statistiques descriptives sur la variable Q1_val. Alors, on fait:

```
import scipy.stats as stat
stat_des = stat.describe(mydata['Q1_val'])
print(stat_des) # on trouve DescribeResult(nobs=26, minmax=(1, 4), mean=2.6923076923076925, variance=1.0215384615384615, skewness=-0.30297924136006316, kurtosis=-0.9341704166061828)
```

On voit que stat_des est un objet présentant sous la forme d'un tuple qui contient quelques stats (nobs, min max, mean, variance skew, kurt).

D'abord à partir de là, on peut vouloir récupérer individuellement les valeurs. Dès lors, il faut spécifier les indices. Exemple:

```
nobs_Q1_val=stat_des[0]
mean_Q1_val=stat_des[2]
variance_Q1_val=stat_des[3]
```

Cependant pour récupérer le min et le max, on procède un peu différemment car ces valeurs sont contenues dans un autre tuple. On a là un tuple de tuples.

```
min_Q1_val=stat_des[1][0]
max_Q1_val=stat_des[1][1]
print(min_Q1_val, max_Q1_val)
```

On pouvait aussi faire une assignation multiple lors du calcul des statistiques descriptves. Exemple:

```
nobs_Q1_val, min_max, mean_Q1_val, variance_Q1_val, skew , kurt=stat.describe(mydata['Q1_val'])
```

Cela permet de distribuer le contenu de l'objet aux différents objets indiqués.

Mais puisque min_max est un tuple. Pour calculer le min et le max, on fait:

```
min_Q1_val=min_max[0]
max_Q1_val=min_max[1]
```

Une autre manière plus directe pour réaliser les statistiques descriptives est d'utiliser la fonction describe() associée à une table dataframe. Par exemple:

```
x=mydata['Q1_val'].describe()
print(x)
```

Cette fonction crée un objet dataframe pour stocker les statistiques. Les indices de cette dataframe sont justement les noms des statstiques

On peut utiliser cette méthode describle() à toute la table. Pour cela, les stats sont produites uniquement pour les variables quantitatives. Exemple:

```
x=mydata.describe()
print(x)
```

5.2.1.2. Utilisation du module statsmodels

```
import statsmodels
import statsmodels.stats.weightstats as stat
x=stat.DescrStatsW(mydata['Q1_val']) # définition de l'objet contenant les statistiques descriptives
print(dir(x)) # pour voir l'ensemble des méthodes associée
print(x.mean) # Affichage de la moyenne
print(x.std)  # Affichage de l'ecart-type
print(x.nobs) # Affichage du nombre d'obs
```

Par ailleurs, il faut noter que dans toute variable quantiative dans une table, il ya des méthodes associées telles que: min(), max(), mean(), etc. On peut appliquer ces méthodes pour renvoyer les statistiques. Exemple:

```
stat=[mydata['Q1_val'].min(),mydata['Q1_val'].max(),mydata['Q1_val'].mean(),(mydata['Q1_val'].std())**2,mydata['Q1_val'].median(),mydata['Q1_val'].skew(),mydata['Q1_val'].kurt()]
print(stat) # attention: il semble qu'il y ait de legère différence dans certaines valeurs kurt et skew.
```

On peut également utiliser la function numpy pour calculer les indicateurs de base. Par exemple:

```
mean_Q1_val=numpy.mean(mydata['Q1_val'])
median_Q1_val=numpy.median(mydata['Q1_val'])
print(median_Q1_val)
```

Tout ceci montre donc qu'il ya plusieurs approches pour calculer les statistiques descriptives.

5.2.2. Statistiques descriptives sur les variables qualitatives

5.2.2.1. Tableau de fréquence univarié (tri simple)

Pour faire le tableau de fréquence simple on peut utiliser la fonction values_counts() de pandas.

Exemple: Soit la table de donnée suivante:

```
mydata=pandas.read_excel("QUESTIONNAIRE_ACCEUIL.xlsx",sheetname="Feuil1", header=0, parse_cols="A:R")
print(mydata.head(n=5))
```

Faisons le tableau de fréquence de la variable Q1_mod. On a:

```
x=pandas.value_counts(mydata['Q1_mod'])
```

```
print(x)
```

Pour présenter les résultats sous forme de pourcentage, il faut effectuer des calculs supplementaires:

```
dico=dict(x) # Transformer x en un dictionnaire
print(dico)
```

Transformation du dictionnaire en dataframe

```
col=dico.keys()   # récupérer les clés du dictionnaires (Ces clés représentent les modalités analysées. Elles seront les variables dans le dataframe à créer).
print(col)
df = pandas.DataFrame(dico,columns=col, index=[0]) # index=[0] doit être spécifié lorsque les valeurs du dictionnaire ne sont pas encadré par [].
df['total']=df.sum(axis=1)
for names, values in df.iteritems(): # on récupère le nom de la colonne et sa valeur
    df['{name}'.format(name=names)]
=100*df['{name}'.format(name=names)]/df['total']    #    Calcul    du pourcentage
print(df) # on obtient ainsi le tableau de fréquence en pourcentage
```

5.2.2.2. Tableau de fréquences croisé

Soit la base de donnée suivante:

```
df=pandas.read_csv("binary.csv",sep=',', header=0, encoding ='utf-8')
# ou encoding ='Latin-1'
print(df.head(n=5))
x=pandas.crosstab(df['admit'], df['rank'], rownames=['admit']) # à revoir
print(x) # tableau croisé entre admit et rank
```

Ce tableau donne les fréquences absolues. Pour avoir les fréquences relatives, on a:

```
# Pourcentage par rapport à la ligne
x=pandas.crosstab(df['admit'], df['rank']).apply(lambda r: r/r.sum(), axis=1)
print(x)
# Pourcentage par rapport à la colonne
x=pandas.crosstab(df['admit'], df['rank']).apply(lambda r: r/r.sum(), axis=0)
print(x)
# Pourcentage par rapport au total
x=pandas.crosstab(df['admit'], df['rank']).apply(lambda r: r/len(df), axis=0)
print(x)
```

5.2.3. Statistiques descriptives croisées et mesures d'associations

5.2.3.1. Mesure d'assocation entre variables quantitatives : le coefficient de corrélation

Pour mesurer la correlation entre deux variables quantitatives, on peut utiliser la fonction pearsonr() de scipy.stats. Toutefois, cette fonction n'est applicable que dans le cas de deux variables. Pour le cas de plusieurs variables, il faut utiliser les fonctionnalités des modules comme numpy ou statsmodels. Ci-dessous quelques exemples.

Corrélation entre deux variables

Exemple: Calculons la corrélation entre entre les variables Q3 et Q4 de la table de données mydata. On a:

```
import scipy.stats as stat
print(stat.pearsonr(mydata['Q3'],      mydata['Q4']))      #      (-
0.21175438021361556, 0.29905014567129801) coefficient et pvalue test
significativité.
```

Corrélation entre plusieurs variables

Malheureusement la fonction pearsonr() ne permet pas de calculer la matrice de corrélation lorqu'il s'agit de plusieurs variables. C'est pourquoi, il faut utiliser d'autres modules comme numpy ou statsmodels, etc.

Exemple: Calcul de la matrice de corrélation entre Q3, Q4, Q5 et Q6 :

Utilisation de la fonction corrcoef() de statsmodels

import statsmodels.stats.weightstats as stat

```
x=stat.DescrStatsW(mydata[['Q3','Q4','Q5','Q6']])   #   définition   de
l'objet contenant les statistiques descriptives
print(x.corrcoef) # matrice de corrélation
print(dir(x)) # pour voir l'ensemble des méthodes associée
```

Utilisation de la fonction corrcoef de numpy

Construisons un array à partir d'une liste des listes formées des variables

```
x=numpy.array([mydata['Q3'] ,mydata['Q4'] ,mydata['Q5'],mydata['Q6']])
print(numpy.corrcoef(x))
```

On obtient ainsi un array qui contient les coefficients de corrélation dans l'ordre indiqué des variables.

NB : On pouvait également utiliser le module matplotlib pour affichier les coefficients de corrélation

```
import matplotlib.pyplot as plt
x=mydata[['Q3','Q4','Q5','Q6']]
print(plt.matshow(x.corr()))
plt.show()
#plt.cla()
#plt.clf()
plt.close()
```

5.2.3.2. Mesure d’association entre variables qualitatives : le test d'indépendance de khi-deux

Exemple :

Soit le tableau de fréquence suivant (tableau de contingence 2x3):

```
obs = numpy.array([[10, 10, 20], [20, 20, 20]])
Le test de khi-deux se met en œuvre comme suit :
import scipy.stats as stat
print(stat.chi2_contingency(obs))        #        (2.7777777777777777,
0.24935220877729619,  2,  array([[  12.,    12.,    16.],  [  18.,    18.,
24.]])) stat khi-deux, pvalue, ddl, et tableau de fréquence théorique.
```

Lorsque le nombre d'observations dans les cellules sont faibles (<5), on utilise le test exact de fisher avec la fonction scipy.stats.fisher_exact

Exemple:

Soit le tableau de fréquence suivant:

```
obs = numpy.array([[8, 2], [1, 5]])
Le test exact de Fisher
import scipy.stats as stat
print(stat.fisher_exact(obs)) # (20.0, 0.034965034965034919) oddsratio
et pvalue
```

5.3. Calcul de quantiles, de fonction de répartition et de fonction de densité

5.3.1. Fonction de répartition et calcul de quantile

On peut calculer les fonctions de répartition empirique d'une variable en utilisant la méthode percentileofscore() de la fonction scipy.stats. Par exemple on sait que la médiane a une fonction de répartition égale à 50%. Pour le vérifier prenons la valeur de la médiane calculée et appliquons percentileofscore().

```
import scipy.stats as stat
median_val=[mydata['Q1_val'].median()]
print(stat.percentileofscore(mydata['Q1_val'],median_val,kind='mean'))
```

5.3.2. Lecture des tables statistiques avec python

On veut obtenir les valeurs des quantiles et des fonctions de répartition pour différentes lois statistiques utilisées pour l'inférence.

5.3.2.1. Loi normale centrée et réduite

```
import scipy.stats as stat
# Test bilatéral: calcul de la stat de test avec alpha= 0.05
print(stat.norm.ppf(0.975,loc=0,scale=1)) # 1.96
# Test bilatéral: calcul de la pvalue avec alpha= 0.05
print(stat.norm.cdf(abs(1.96),loc=0,scale=1)) # 0.975 ( cette valeur
est égale 1-pvalue/2)
# Test unilatéral à droite: calcul de la stat de test avec alpha= 0.05
print(stat.norm.ppf(0.95,loc=0,scale=1)) #  1.64485
# Test unilatéral à droite: calcul de la p-value
print(stat.norm.cdf(1.64485,loc=0,scale=1)) # 0.95 ( cette valeur est
égale 1-pvalue)
# Test unilatéral à gauche: calcul de la stat des test avec alpha=
0.05
print(stat.norm.ppf(0.05,loc=0,scale=1)) #  -1.64485
# Test unilatéral à gauche: calcul de la pvalue
print(stat.norm.cdf(-1.64485,loc=0,scale=1)) # 0.05 ( cette valeur est
égale pvalue)
```

5.3.2.2. Loi de Student

Exemple : avec ddl = 500

```
import scipy.stats as stat
# Test bilatéral: calcul de la stat de test avec alpha= 0.05
print(stat.t.ppf(0.975,df=500)) # 1.96471983747 (la loi de student
converge vers la loi normale lorsque le df est plus élevé).
# Calcul de la p-value
print(stat.t.cdf(abs(1.96471983747),df=500)) # 0.975 ( cette valeur
est égale 1-pvalue/2)
# Test unilateral à droite avec alpha= 0.05
print(stat.t.ppf(0.95,df=500)) # 1.64790685393 (la loi de student
converge vers la loi normale lorsque le df est plus élevé).
# Calcul de la p-value
print(stat.t.cdf(1.64790685393,df=500)) # 0.95 ( cette valeur est
égale 1-pvalue)
# Test unilateral à gauche avec alpha= 0.05
print(stat.t.ppf(0.05,df=500)) # -1.64790685393 (la loi de student
converge vers la loi normale lorsque le df est plus élevé).
```

```
# Calcul de la p-value
print(stat.t.cdf(-1.64790685393,df=500))  #  0.05  (  cette  valeur  est
égale pvalue)
```

5.3.2.3. Loi du khi-2

Exemple : avec ddl = 10

```
import scipy.stats as stat
Calcul de la statistique de test avec alpha 0.05
print(stat.chi.ppf(0.95,df=10)) # 4.27867246389
# Calcul de la p-value
print(stat.chi.cdf(4.27867246389,df=10))  #  0.95(  cette  valeur  est
égale 1-pvalue)
```

5.3.2.4. Loi de Fisher

Exemple avec ddl numérateur = 1, ddl dénominateur = 30

```
import scipy.stats as stat
Calcul de la statistique de test avec alpha 0.05
print(stat.f.ppf(0.95,dfn=1,dfd=30)) # 4.17087678577
# Calcul de la p-value
print(stat.f.cdf(4.17087678577,dfn=1,dfd=30))  #  0.95  #  cette  valeur
est égale à 1-pvalue
```

5.4. Tests d'adéquation à une Loi

Le test d'adéquation à une loi donnée consiste à tester la distribution d'une variable suit la loi en question. Il existe plusieurs fonctions pour tester l'adéquation à une loi notamment celles du module scipy ou du module Statsmodels

5.4.1. Test d'adéquation à une loi normale

Exemple : Considérons la variable Q1_val de la table mydata et testons la normalité de cette variable.

5.4.1.1. Test de normalité d'Agostino

```
import scipy.stats as stat # Utilisation du module scipy
ag = stat.normaltest(mydata['Q1_val']) # message d'avertissement, si n
est trop faible pour un test fiable
print(ag) # (2.0091276938071805, 0.36620432112621987) statistique de
test et p-value , h0: normalité;  (si p-value < α, rejet de l'hyp. de
normalité). ici, on accepte
```

5.4.1.2. Test de Normalité Shapiro-Wilks

```
import scipy.stats as stat # Utilisation du module scipy
sp = stat.shapiro(mydata['Q1_val'])
print(sp) # ((0.8732276558876038, 0.004154810681939125), statistique de test et p-value , h0: non normalité;  (si p-value < α, acceptation de l’hyp. de normalité). ici, on accepte
```

5.4.1.3. Test de normalité d'Anderson-Darling

```
import scipy.stats as stat # Utilisation du module scipy
ad = stat.anderson(mydata['Q1_val'],dist="norm") # test possible pour autre loi que « norm »
print(ad) # (1.2559218397601803, array([ 0.516, 0.587, 0.705, 0.822, 0.978]), array([ 15. , 10. , 5. , 2.5, 1. ]))
```

La stat de test, seuils critiques pour chaque niveau de risque, on constate ici que la p-value est sup. à 15%.

Utilisation du module statsmodels

```
import statsmodels.stats.diagnostic as stat
print(stat.normal_ad(mydata['Q1_val'])) # (1.2559218397601803, 0.0022978200411569029) stat et pvalue.
```

5.4.1.4. Test de normalité de Kolmogorov-Smirnov

```
import scipy.stats as stat # Utilisation du module scipy
print(stat.kstest(mydata['Q1_val'], 'norm')) # (0.84134474606854293, 0.0) stat et pvalue.
```

5.4.1.5. Le test de normalité de Lillifors

```
import statsmodels.stats.diagnostic as stat # Utilisation du module statsmodels
print(stat.kstest_normal(mydata['Q1_val'], pvalmethod='approx'))
```

5.4.2. Test d'adéquation à une loi de khi-deux

Exemple : soit x=[16, 18, 16, 14, 12, 12] les fréquences observées d'une variable qualitative. On peut tester l'adéquation à la loi de khi-deux comme suit:

```
f_obs =[16, 18, 16, 14, 12, 12]  # f_obs . Généralement cette table est obtenue à partir d’un tableau de fréquences (voir section sur tableau de fréquence)
print(stat.chisquare(f_obs)) # 2.0, 0.84914503608460956) stat et pvalue.
```

Dans l’exemple ci-dessus, les fréquences théoriques sont supposées suivre une loi uniforme. Supposons maintenant que les fréquences théoriques soitent connues telle que , le test se présente comme suit:

```
f_obs =[16, 18, 16, 14, 12, 12]
f_theo=[16, 16, 16, 16, 16, 8]
```

Le test d'adéquation se met œuvre alors comme suit:

```
print(stat.chisquare(f_obs , f_exp=f_theo)) # (3.5,
0.62338762774958223)
```

Il faut noter qu'on peut calculer les fréquences théoriques en utilisant la fonction scipy.stats.contingency.expected_freq

5.5. Test de conformité à valeur de référence : test d'égalité de la moyenne à une valeur

Considérons la variable mydata['Q1_val']. On va tester si sa moyenne est égale à 3. Il s'agit d'un test de conformité de la moyenne. Alors on fait:

```
import scipy.stats as stat
print(stat.ttest_1samp(mydata['Q1_val'],popmean=3)) # (-
1.5523010514126647, 0.1331589799069999) statistique de test et pvalue.
Ici on accepte h0.
```

On pouvait aussi effectuer le calcul manuel comme suit:

1-Calcul de la moyenne et de l'écart-type empiriques

```
m=m = numpy.mean(mydata['Q1_val'])
sigma=numpy.std(mydata['Q1_val'],ddof=1) # ddof=1 pour calculer la
variance emprique modifié avec N-1
```

2- Calcul de la statistique de test

```
import math
t = (m - 3)/(sigma/math.sqrt(mydata['Q1_val'].size)) # on trouve -
1.55230105141 comme dans le test automatique
```

3- Calcul de la p.value : On calcule la pvalue en utilisant la valeur t et le ddl qui est égale à la size de la variable

```
print(t)
p=stat.t.cdf(math.fabs(t),df=mydata['Q1_val'].size-1) # On enlève 1
car le sigma a été calculé avec N-1
p=stat.t.cdf(abs(t),df=mydata['Q1_val'].size-1)
print(p) # 0.933420510047( cette valeur est égale 1-pvalue/2)
```

La valeur de la pvalue:

```
pvalue=2.0*(1.0 - p)
print(t,pvalue)
```

5.6. Test d'égalité de moyennes sur échantillons indépendants

Soit la table de données suivante qui donne les scores de

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy',
'Kader', 'Jean' , 'Philipe', 'Paul', 'Pierre', 'Jacques'],
        'pre_score': [4, 24, 31, 2, 3, 4 , 10 , 25 , 40, 30, 18],
        'mid_score': [25, 94, 57, 62, 70, 12 , 24 , 25 ,64, 28, 20],
        'post_score': [5, 43, 23, 23, 51, 14 , 27 , 26, 70, 80 ,30],
        'Area': ['Rural', 'Urban', 'Rural', 'Urban', 'Urban', 'Rural',
'Urban' , 'Rural', 'Urban', 'Urban', 'Rural']}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score',
'mid_score', 'post_score', 'Area'])
print(df)
```

On souhaite comparer la moyenne de la variable pre_score entre le milieu urbain et le milieu rural. Pour cela on va utiliser la fonction scipy.stats.ttest_ind. On a:

```
import scipy.stats as stat
datar=df[df['Area']=='Rural']['pre_score']
datau=df[df['Area']=='Urban']['pre_score']
print(stat.ttest_ind(datar,      datau,      equal_var=True))      #(-
0.20595733971285773, 0.84140772015831655) t-stat et pvalue. On accepte
l'égalité
```

Effectuons le test pour post_score

```
print(stat.ttest_ind(df[df['Area']=='Rural']['post_score'],
df[df['Area']=='Urban']['post_score'],    equal_var=True))    #    (-
2.6561445957935592, 0.026210890486496038) on rejette l'égalité
```

5.7. Test d'égalité de moyennes sur échantillons indépendants appariés

Soit la table de données suivante qui donne les scores de 11 individus

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy',
'Kader', 'Jean' , 'Philipe', 'Paul', 'Pierre', 'Jacques'],
        'pre_score': [4, 24, 31, 2, 3, 4 , 10 , 25 , 40, 30, 18],
        'mid_score': [25, 94, 57, 62, 70, 12 , 24 , 25 ,64, 28, 20],
        'post_score': [5, 43, 23, 23, 51, 14 , 27 , 26, 70, 80 ,30],
        'Area': ['Rural', 'Urban', 'Rural', 'Urban', 'Urban', 'Rural',
'Urban' , 'Rural', 'Urban', 'Urban', 'Rural']}
```

```
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score',
'mid_score', 'post_score', 'Area'])
print(df)
```

On souhaite comparer la moyenne des variables pre_score et post_score qui son les scores mesurés sur les mêmes individus mais à des dates différentes. Pour cela on va utiliser la fonction scipy.stats.ttest_rel. On a:

```
import scipy.stats as stat
print(stat.ttest_rel(df['pre_score'],    df['post_score']))    #    (-
3.2712562972589461, 0.0084124074507500833) t-stat et pvalue
```

5.8. Comparaison de moyennes sur plusieurs groupes : le test ANOVA

L'anova est un test de comparaison de moyenne entre deux ou plusiers groupes basé sur l'hypothèse de normalité. Pour le réaliser on utilise la fonction scipy.stats.f_oneway.

Exemple: Faisons une anova sur le pre_score entre le milieu urabin et le milieu rural

```
datar=df[df['Area']=='Rural']['pre_score']
datau=df[df['Area']=='Urban']['pre_score']
import scipy.stats as stat
print(stat.f_oneway(datar,     datau))     #     0.042418425781597317,
0.84140772015831566) F-stat et pvalue
```

5.9. Modèles de régressions linéaires multiples

Les modèles de régressions linéaires multiples visent à mesurer l'influence de plusieurs variables explicatives sur une variable expliquée. Pour mettre en œuvre ces méthodes sous python, nous allons nous servir soit du module Statsmodels, soit du module sklearn.

5.9.1. Estimation d'un modèle de régression linéaire

5.9.1.1. Utilisation du module statsmodels

Soit la table de données suivante.

```
df=pandas.read_excel("boston hdata.xlsx",sheetname="boston", header=0,
parse_cols="A:N")
print(df.head(n=5))
```

On souhaite expliquer le prix des loyers MEDV (variable y) par un ensemble de variables explicatives : CRIM, ZN, INDUS, CHAS , NOX , RM , AGE , DIS , RAD , TAX , PTRATIO , B , LSTAT (voir la description des variables dans le fichier excel).

Préparation de la table de donées pour la régression :

Nb : pour l'estimation du modèle MCO avec le module statsmodels il faut ajouter une colonne supplémentaire contant uniquement des 1 pour représenter la constante dans le modèle. Ainsi on a:

```
df['Const'] = numpy.ones(( len(df), )) # ajout d'une colonne supplémenatire remplie de 1.
x = df.drop(['MEDV'], axis=1) # Construction des variables explicatives en supprimant la colonne MEDV
print(x)
y=df['MEDV'] # variable y
# estimation du modèle
import statsmodels.formula.api as sm
regr = sm.OLS( y, x ).fit() # estimation
print(regr.summary()) # affichage des résultats (on voit que l'affichage est beaucoup plus meilleur en statsmodels)
```

On pouvait aussi utiliser la spécification en formule comme suit:

```
import statsmodels.formula.api as smf
regr = smf.ols(formula='MEDV ~ CRIM + ZN + INDUS + CHAS + NOX + RM + AGE + DIS + RAD + TAX + PTRATIO + B + LSTAT', data=df).fit()
print(regr.summary())
```

Pour estimer le modèle sans constante on ajoute -1 à la formule (si la colonne remplie de 1 a été déjà ajoutée). Sinon, on estime simplement le modèle.

```
import statsmodels.formula.api as smf
regr = smf.ols(formula='MEDV ~ CRIM + ZN + INDUS + CHAS + NOX + RM + AGE + DIS + RAD + TAX + PTRATIO + B + LSTAT -1', data=df).fit()
print(regr.summary())
```

Traitement des variables catégorielles dans la régression

Cas d'une variable catégorielle de type caractère

Exemple : soit la table

```
df=pandas.read_csv("SAheart.csv",sep=';', header=0, encoding ='utf-8') # ou encoding ='Latin-1'
```

On va regresser la variable chd sur les variable: sbp tobacco ldl adiposity famhist typea obesity alcohol age.

Mais la variable famhist est une variable catégorielle de type caractères. On a donc plusieurs méthodes pour traiter cette variable :

Première méthode: Intégrer directement utiliser la variable dans la formule en utilisant le module statsmodels:

```
import statsmodels.formula.api as smf
```

```
regr = smf.ols(formula='chd ~ sbp + tobacco +  ldl + adiposity +
famhist + typea + obesity + alcohol +age', data=df).fit()
print(regr.summary())
```

Deuxième méthode : recoder la variable en numérique et utiliser la fonction c()

```
df['famhist_cat'] = pandas.Categorical(df['famhist']).labels #
Transforme la variable famhist en variable catégorielle numérique
import statsmodels.formula.api as smf
#regr = smf.ols(formula='chd ~ sbp + tobacco +  ldl + adiposity +
c(famhist_cat) + typea + obesity + alcohol +age', data=df).fit() #
Fonction c() à revoir
print(regr.summary())
```

Cas d'une variable catégorielle de type numérique

Comme discuté ci-dessus, pour inclure une variable catégorielle de type numérique dans l'estimation, on utilise simplement la fonction c()

```
import statsmodels.formula.api as smf
regr = smf.ols(formula='chd ~ sbp + tobacco +  ldl + adiposity +
c(famhist_cat) + typea + obesity + alcohol +age', data=df).fit() #
Fonction c() à revoir
print(regr.summary())
```

5.9.1.2. Utilisation du module sklearn

Le module sklearn qui est un module spécialisé dans le machine-learning offre aussi des alternatives au module statsmodels pour l'estimation d'un modèle de régression linéaire.

Toutefois, il faut noter que ce module nécessite que les données soient mis sous le format vertical (voir chapitre 3 pour plus de détails sur la mise en format vertical des données). Ici, on utilise la fonction numpy.matrix() pour réaliser cette tâche.

Exemple :

Soit la table de données suivante.

```
df=pandas.read_excel("boston hdata.xlsx",sheetname="boston", header=0,
parse_cols="A:N")
print(df.head(n=5))
data = numpy.matrix( df ) # convertir en format vertical
#  extraction de la variable x et y
y= data[:,-1] # la variable  MEDV se trouve sur la dernière colonne
# y= df['MEDV'] # la variable  MEDV se trouve sur la dernière colonne
(equivalent: pas besoin de convertir en format verticale car une seule
dimension)
```

```
x = data[:,:-1] # Sélectionner toutes les variable  sauf la dernière colonne (NB: Si la variable n'était pas sur la dernière colonne on allait faire la méthode suivante:)
##x =df.drop(['MEDV'], axis=1)# on garde toutes les autres variables restantes comme explicatives (puisque la variable CHAS se trouve en milieu de tableau, on ne peut pas directement indicer)
##x = numpy.matrix( x ) # Ensuite, on convertit en format vertical
print(x)
# préparation du modèle
from sklearn import linear_model
regr = linear_model.LinearRegression(fit_intercept=True) # définition du modèle (avec constante)
regr.fit(x, y)
print('Coefficients: \n', regr.coef_) # coefficients
print ('Constante', regr.intercept_) # la constante
print('R²: %.2f' % regr.score(x, y)) # R2 (# Explained variance score)
```

5.9.2. Analyse prédictive à partir du modele de régression linéaire avec le module sklearn

5.9.2.1. Estimation et validation du modèle

Dans cette section, nous allons estimer un modèle de régression linéaire (sur un échantillon d'apprentissage), ensuite le valider sur un échantillon de validation (backtesting). Pour cela, nous allons utiliser la base de données diabetes disponible dans la documentation de python.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
import sklearn
# Charger la base de donnée diabète
diabetes = datasets.load_diabetes()
print(diabetes)
```

C'est une base de données qui contient deux sous-bases de données: la première nommée target et la seconde nommée data. target est un array à une dimension (formée d'une seule liste) alors que data est un array à plusieurs dimensions (formée de plusieurs listes).

Pour effectuer les analyses, nous allons extraire les éléments de chaque base de donnée.

Pour l'array data, nous allons allons extraire le troisième élement de chaque liste qui le compose pour former la base diabetes_X

```
diabetes_X = diabetes.data[:, np.newaxis, 2] # Extraction d'un array à plusieurs dimensions (où chaque liste contient qu'un seul élements égal au troisième élément des listes intiales)
print(diabetes_X) # Notons ici que les données sont déjà sous format verticale.
```

Nous allons scinder cette base en un échantillon apprentissage et un échantillon de validation

```
diabetes_X_train = diabetes_X[:-20] # Echantillon d'apprentissage (sélectionner tous les éléments avant le 20ième élement à partir de la gauche (exclu))
print('diabetes_X_train')
print(diabetes_X_train)
diabetes_X_test = diabetes_X[-20:] # Echantillon de validation (sélectionner tous les élement après à partir du 20ième élement à partir de la gauche  (inclusiement))
print(type(diabetes_X_test))
```

Scinder la table target en échantillon d'apprentissage et de validation de la même dimension que la table diabetes_X_train et diabetes_X_test

```
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test = diabetes.target[-20:]
# Préparation du modèle de régression
from sklearn import datasets, linear_model
import sklearn
regr = linear_model.LinearRegression()
```

Estimation du modèle sur l'échantillon d'apprentissage avec comme variable explicative diabetes_X_train et variable expliquée diabetes_y_train

```
regr.fit(diabetes_X_train, diabetes_y_train)
print('Coefficients: \n', regr.coef_) # coefficients
print ('intercept', regr.intercept_) # la constante
print("Residual sum of squares: %.2f" % np.mean((regr.predict(diabetes_X_test) - diabetes_y_test) ** 2)) # Erreur quadratique moyenne
print('Variance score: %.2f' % regr.score(diabetes_X_test, diabetes_y_test)) # R2 (# Explained variance score: 1 is perfect prediction)
```

Examen graphique de la qualité du modèle

```
plt.scatter(diabetes_X_test, diabetes_y_test, color='black') # Représentation sur l'échantillon d'apprentissage
plt.plot(diabetes_X_test, regr.predict(diabetes_X_test), color='blue', linewidth=3) # Représentation de la valeur prédite en fonction diabetes_X_test
plt.xticks(())
plt.yticks(())
plt.show()
plt.cla()
plt.clf()
plt.close()
```

5.10. Régression logistique binaire

Le modèle de regression linéaire vise à expliquer la probabilité d'un évènement (captée par une variable binaire codée 1 et 0) par un ensemble de variables explicatives.

5.10.1. Estimation du modèle logistique binaire

Comme pour le modèle de régression linéaire, on peut utiliser soit le module statsmodels ou le module sklearn

5.10.1.1. Utilisation du module statsmodels

Pour effectuer la régression logit on utilise la fonction linear_model.LogisticRegression().

Exemple :

Soit la base de données suivante:

```
df=pandas.read_csv("binary.csv",sep=',', header=0, encoding ='utf-8')
# ou encoding ='Latin-1'
print(df.head(n=5))
```

La table contient les variables admit,gre,gpa,rank. On veut régresser admit (0, 1) sur les variables gre,gpa et rank. On va renommer la variable rank en prestige pour eviter les conflits de nom avec les functions de python

```
df.rename(columns={'rank': 'prestige'}, inplace=True)
```

Ensuite, on va découper la variable prestige en variables binaires

```
dummy_ranks = pandas.get_dummies(df['prestige'], prefix='prestige')
```

On va ajouter ces variables binaires aux données initiales

```
df = pandas.concat([df, dummy_ranks], axis=1)
```

On va ensuite ajouter à la table une variable représentant la constante (remplie de 1)

```
df['Const'] = numpy.ones(( len(df), ))
On va extraire la variable expliquée
y=df['admit'] # variable y
```

Ensuite extraction des variables explicatives

```
x = df.drop(['admit', 'prestige','prestige_1'], axis=1) # on exclut ces variables pour retenir les restant. NB:prestige_1 est la modalité de référence.
print(x)
```

Préparation de l'estimation du modèle

```
import statsmodels.api as sm
import statsmodels.discrete.discrete_model as dm
result = dm.Logit(y, x).fit() # Estimation du modèle. on peut aussi utilise sm.
print(result.summary())
```

Pour présenter séparémment les intervalles de confiance, on fait:

```
print(result.conf_int())
```

Pour obtnir les odds ratio, on fait:

```
odds=numpy.exp(result.params)
print(odds)
```

Pour obtenir l'intervalle de confiance des odd-ratio, on fait:

```
odds_ci=numpy.exp(result.conf_int())
print(odds_ci)
```

Pour calculer les probabilités prédites (sur l'échantillon d'apprentissage)

```
df['admit_predprob'] = result.predict(x)
```

Pour calculer les prédictions linéaires

```
df['admit_predlin'] = result.predict(x, linear=True)
print(df)
```

Pour estimer le modèle à partir d'une formule, on adopte la spécification suivante:

```
import statsmodels.api as sm
sm.Logit.from_formula(formula='admit ~ gre+ gpa + prestige_2 + prestige_3 + prestige_4', data=df)
```

5.10.1.2. Utilisation du module sklearn

L'utilisation du module sklearn nécessite que les données doivent soient mises sous forme verticale. Nous utilisons donc fonction numpy.matrix pour cette conversion.

Exemple :

```
Soit la table de données suivante.
df=pandas.read_excel("boston hdata.xlsx",sheetname="boston", header=0, parse_cols="A:N")
print(df.head(n=5))
data = numpy.matrix( df ) # Conversion en format verticale
```

```
y= data[:,3] # extraction de la variable CHAS ( 4ième position); format vertical tirée de la forme matricielle. la variable  CHAS est une variable binaire qui sera utilisée comme vd.
# y= df['CHAS']# On pouvait aussi directement extraire la variable CHAS à partir de df sans format verticale ( vvecteur à une seule dimension)
print(y)
x =df.drop(['CHAS'], axis=1)# on garde toutes les autres variables restantes comme explicatives (puisque la variable CHAS se trouve en milieu de tableau, on ne peut pas directement indicer)
x = numpy.matrix( x ) # Ensuite, on convertit en format vertical(valable que dans le cas du OLS)
print(x)
# préparation du modèle
from sklearn import linear_model
# Préparation et estimation du modèle
regr = linear_model.LogisticRegression().fit(x,y)
print( dir(regr))
#help(regr)
# affichage des coefficients
print('Coefficients: \n', regr.coef_) # coefficients
print ('intercept', regr.intercept_) # la constante
```

5.10.2. Selection automatique des variables explicatives dans la regresssion logistique binaire

L'objectif de la sélection automatique des variables explicatives ets de réduire le modèle aux variables explicatives les plus pertinentes

Ici, nous allons utiliser la base de données diabetes pour prédire la probabalité du dibabète en fonction des antécédents cliniques en utilisant le logit et la méthode RFE de sklearn

La sélection de variables - la recherche de modèles parcimonieux - présente plusieurs avantages : interprétation, déploiement (moins de var. à renseigner),

Remarque: nous implémentons la méthode RFE de scikit-learn (recursive feature elimination) qui élimine au fur et à mesure les poids les plus faibles en valeur absolue et s'arrête quand on arrive à la moitié ou à un nombre spécifié de variables.

Attention : les variables ne sont pas toujours à la même échelle, une standardisation des variables paraît nécessaire).

Soit la table de données

```
df=pandas.read_excel("diabetes.xlsx",sheetname="data",        header=0, parse_cols="A:I")
print(df.head(n=5))
# Y : variable cible (diabète)
```

```
# Liste initiale des variables : pregnant, diastolic, triceps, bodymass, pedigree, age, plasma, serum.
#transformation en matrice numpy (format verticale) - seul reconnu par scikit-learn
data=numpy.matrix(df) # première méthode
# data = df.as_matrix() # seconde méthode
X = data[:,0:8] # X matrice des varibale explicatives les 8 première variables
y = data[:,8] #y vecteur de la variable à prédire
#utilisation du module cross_validation de scikit-learn (sklearn) pour subdiviser l'échantillon en un échantillon d'apprentissage et de validation
from sklearn import cross_validation
#subdivision des données – éch.test = 300 ; éch.app = 768 – éch.test = 468
X_app,X_test,y_app,y_test = cross_validation.train_test_split(X,y,test_size = 300,random_state=0)
print(X_app.shape,X_test.shape,y_app.shape,y_test.shape)
#importer la classe LogisticRegression
from sklearn.linear_model import LogisticRegression
#création d'une instance de la classe
lr = LogisticRegression()
#algorithme de sélection de var.
from sklearn.feature_selection import RFE
selecteur = RFE(estimator=lr)
#lancer la recherche
sol = selecteur.fit(X_app,y_app)
#nombre de var. sélectionnées
print(sol.n_features_) # 4 � 4 = 8 / 2 variables sélectionnées
#liste des variables sélectionnées
print(sol.support_) # [True False False True True False True False ]
#ordre de suppression
print(sol.ranking_) # [1 2 4 1 1 3 1 5]
#réduction de la base d'app. aux var. sélectionnées
#en utilisant le filtre booléen sol.support_
X_new_app = X_app[:,sol.support_]
print(X_new_app.shape) # (468, 4) � 4 variables restantes
#construction du modèle sur les explicatives sélectionnées
modele_sel = lr.fit(X_new_app,y_app)
#réduction de la base test aux mêmes variables
X_new_test = X_test[:,sol.support_]
print(X_new_test.shape) # (300, 4)
#prédiction du modèle réduit sur l'éch. test
y_pred_sel = modele_sel.predict(X_new_test)
#évaluation
from sklearn import metrics
print(metrics.accuracy_score(y_test,y_pred_sel)) # 0.787 # Aussi bien (presque, 0.793) que le modèle initial, mais avec moitié moins de variables.
```

5.10.3. Analyse prédictive à partir du modèle logistique binaire avec le module sklearn

Ici, nous allons utiliser la base de données diabetes pour prédire la probabalité du dibabètes en fonction des antécédents cliniques

Soit la table de données

```
df=pandas.read_excel("diabetes.xlsx",sheetname="data",        header=0,
parse_cols="A:I")
print(df.head(n=5))
# Y : variable cible (diabète)
#transformation en matrice numpy (format verticale) - seul reconnu par
scikit-learn
# data=numpy.matrix(df)
data = df.as_matrix()# seconde méthode
X = data[:,0:8] # X matrice des varibale explicatives les 8 première
variables
y = data[:,8] #y vecteur de la variable à prédire
#utilisation du module cross_validation de scikit-learn (sklearn) pour
subdiviser l'échantillon en un échantillon d'apprentissage et de
validation
from sklearn import cross_validation
#subdivision des données – éch.test = 300 ; éch.app = 768 – éch.test =
468
X_app,X_test,y_app,y_test                                             =
cross_validation.train_test_split(X,y,test_size = 300,random_state=0)
print(X_app.shape,X_test.shape,y_app.shape,y_test.shape)
#importer la classe LogisticRegression
from sklearn.linear_model import LogisticRegression
#création d'une instance de la classe
lr = LogisticRegression()
#exécution de l'instance sur les données d'apprentissage c.à-d.
construction du modèle prédictif
modele = lr.fit(X_app,y_app)
#les sorties
print(modele.coef_,modele.intercept_)
```

Attention : la régression logistique de scikit-learn s'appuie sur un algorithme différent de celui des autres logiciels de statistique.

NB: On ne dispose pas des indicateurs usuels de la régression logistique (tests de significativité, écarts-type des coefficients, etc.)

Etapes de validation

```
#prédiction sur l'échantillon test
y_pred = modele.predict(X_test)
```

```
#importation de metrics - utilisé pour les mesures de performances
from sklearn import metrics
#matrice de confusion: confrontation entre Y observé sur l'échantillon
de test et la prédiction
cm = metrics.confusion_matrix(y_test,y_pred)
print(cm)
#taux de succès
acc = metrics.accuracy_score(y_test,y_pred)
print(acc) # 0.793 = (184 + 54)/ (184 + 17 + 45 + 54)
#taux d'erreur
err = 1.0 - acc
print(err) # 0.206 = 1.0 – 0.793
#sensibilité (ou rappel)
se = metrics.recall_score(y_test,y_pred,pos_label=1)
print(se) # 0.545 = 54 / (45+ 54)
```

Construction de sa propre mesure de performance (exemple. Sensibilité ou Spécificité)

```
#écrire sa propre func. d'éval - ex. specificité
def specificity(y,y_hat):
#matrice de confusion – un objet numpy.ndarray
    mc = metrics.confusion_matrix(y,y_hat)
    #''negative'' est sur l'indice 0 dans la matrice
    import numpy
    res = mc[0,0]/numpy.sum(mc[0,:])
    #retour
    return res
#la rendre utilisable - transformation en objet scorer
specificite = metrics.make_scorer(specificity,greater_is_better=True)
#utilisation de l'objet scorer
#remarque : modele est le modèle élaboré sur l'éch. d'apprentissage
sp = specificite(modele,X_test,y_test)
print(sp) # 0.915 = 184 / (184 + 17)
```

Stratégie et évaluation sur les petits échantillons:VALIDATION CROISÉE

Problème : lorsque l'on traite un petit fichier (en nombre d'obs.), le schéma apprentissage – test est pénalisant (apprentissage : on réduit l'information disponible pour créer le modèle ; test : un faible effectif produit des estimations des performances très instables).

Solution : (1) construire le modèle sur la totalité des données, (2) évaluer les performances à l'aide des techniques de ré-échantillonnage (ex. validation croisée)

Mise en œuvre : Considérons les deux variables X et y qui ont été définies précédemment. On a:

```
#importer la classe LogisticRegression
from sklearn.linear_model import LogisticRegression
#création d'une instance de la classe
```

```
lr = LogisticRegression()
#exécution de l'instance sur la totalité des données (X,y)
modele_all = lr.fit(X,y)
#affichage
print(modele_all.coef_,modele_all.intercept_)
#utilisation du module cross_validation
from sklearn import cross_validation
#évaluation en validation croisée : 10 cross-validation
succes =
cross_validation.cross_val_score(lr,X,y,cv=10,scoring='accuracy')
#détail des itérations
print(succes)
#moyenne des taux de succès = estimation du taux de succès en CV
print(succes.mean()) # 0.767
```

Le premier travail consiste à séparer les échantillons en une partie apprentissage et une autre de test pour estimer sans biais l'erreur de prévision.

L'optimisation (biais-variance) de la complexité des modèles est réalisée en minimisant l'erreur estimée par validation croisée.

5.10.4. Segmentation, scoring et ciblage avec le modele logistique binaire

5.10.4.1. Objectifs généraux

Dans le but de la promotion d'un produit auprès d'un ensemble de clients, l'objectif du ciblage est de contacter le moins de personnes possible pour obtenir le maximum d'achats.

La démarche méthodologique de la mise en œuvre du ciblage est d'attribuer un score aux individus, les trier de manière décroissante de telle sorte qu'un score élevé correspond à une forte appétence au produit. Il s'agit alors d'estimer à l'aide de la courbe de gain le nombre d'achats en fonction d'une taille de cible choisie.

5.10.4.2. Mise en œuvre de la démarche du ciblage

Considérons les deux échantillons précédemment définis dans la section précédente (X_app,y_app) et (X_test,y_test)

```
# estimation du modèle de régression Logistique
from sklearn.linear_model import LogisticRegression
#création d'une instance de la classe
lr = LogisticRegression()
#modélisation sur les données d'apprentissage
modele = lr.fit(X_app,y_app)
#calcul des probas d'affectation sur l'échantillon de test
probas = lr.predict_proba(X_test)
```

```
#score de 'presence'
score = probas[:,1] # [0.86238322 0.21334963 0.15895063 …]
#transf. en 0/1 de Y_test
pos = pandas.get_dummies(y_test).as_matrix()
#on ne récupère que la 2è colonne (indice 1)
pos = pos[:,1] # [ 1 0 0 1 0 0 1 1 …]
#nombre total de positif
import numpy
npos = numpy.sum(pos) # 99 - il y a 99 ind. ‘’positifs’’ dans l’échantillon test.
#index pour tri selon le score croissant
index = numpy.argsort(score) # [ 55 45 265 261 … 11 255 159]
#inverser pour score décroissant – on s’intéresse à forte proba. en priorité
index = index[::-1] # [ 159 255 11 … 261 265 45 55 ]
#tri des individus (des valeurs 0/1)
sort_pos = pos[index] # [ 1 1 1 1 1 0 1 1 …]
#somme cumulée
cpos = numpy.cumsum(sort_pos) # [ 1 2 3 4 5 5 6 7 … 99]
#rappel
rappel = cpos/npos # [ 1/99 2/99 3/99 4/99 5/99 5/99 6/99 7/99 … 99/99]
#nb. obs ech.test
n = y_test.shape[0] # 300, il y a 300 ind. dans l’éch. test
#taille de cible – séquence de valeurs de 1 à 300 avec un pas de 1
taille = numpy.arange(start=1,stop=301,step=1) # [1 2 3 4 5 … 300]
#passer en proportion
taille = taille / n # [ 1/300 2/300 3/300 … 300/300 ]
```

Démarche de construction de la courbe de gain

L’objectif est de réaliser le graphique à partir des vecteurs ‘taille’ et ‘rappel’ – On parle de « courbe de gain » ou « courbe lift cumulée ».

```
#graphique avec matplotlib
import matplotlib.pyplot as plt
#titre et en-têtes plt.title('Courbe de gain')
plt.xlabel('Taille de cible')
plt.ylabel('Rappel')
#limites en abscisse et ordonnée
plt.xlim(0,1)
plt.ylim(0,1) #astuce pour tracer la diagonale
plt.scatter(taille,taille,marker='.',color='blue')
#insertion du couple (taille, rappel)
plt.scatter(taille,rappel,marker='.',color='red') #affichage
plt.show()
plt.cla()
plt.clf()
plt.close()
```

5.11. Estimation de modèle de régression logistique multinomiales

Le modèle de régression logistique multinomiale vise à mesurer l'influence d'un ensemble de variables explicatives sur une variable qualitative multimodale. Il s'agit de mesurer la probabilité de survenue de chaque modalité (par rapport à une modalité prise comme référence).

Exemple :

Soit la base de données suivante:

```
import numpy as np
import statsmodels.api as st
iris = st.datasets.get_rdataset('iris', 'datasets')
```

On va regresser la variable Species sur un ensemble de variables explicatives en utilisant le module statsmodels

```
y = iris.data.Species # récupère la variable y
print( y.head(10))
x = iris.data.ix[:, 0] # récupère les variables x
x = st.add_constant(x, prepend = False) # additionner la colonne constante à x (remplie de 1)
print (x.head(10))
mdl = st.MNLogit(y, x) # spécification du modèle
mdl_fit = mdl.fit() # estimation du modèle
print (mdl_fit.summary()) # affichage de la table de régression
mdl_margeff = mdl_fit.get_margeff() # calcul des effets marginaux
print (mdl_margeff.summary())
```

Chapitre 6 : Data visualisation avec python

Ce chapitre présente les différentes méthodes de construction de graphiques sous python pour la visualisation des données. Tout comme les méthodes d'analyses statistiques descriptives, le choix du type de visualisation graphique dépend de la nature des variables. On distingue deux types de graphiques : les graphiques univariés (réalisés sur une seule variable) et les graphiques croisés (réalisé en croisant plusieurs variables). Pour les graphiques univariés sur variables qualitatives, on choisira par exemple les diagrammes de fréquences (barres de fréquences) et les diagrammes circulaires. Pour les graphiques univariés sur variables quantitatives, on choisira les histogrammes, les box-plots, les barres de moyennes, etc… Et pour les analyses croisées, entre deux variables quantitatives, par exemple on choisira un nuage de points ou une courbe d'évolution, etc.. Tandis que pour une analyse croisée entre une variable quantitative et une variable qualitative, on choisira les histogrammes, les box-plots (ou les barres de moyennes) présentés selon les différentes modalités de la variable qualitative. Ce chapitre vise à présenter brièvement chacun de ces cas.

6.1. Les graphiques à barres

6.1.1. Barres simples de comparaison de moyennes

Soit la table de données suivante qui fournit le score de 5 étudiants à trois tests

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'pre_score': [4, 24, 31, 2, 3],
        'mid_score': [25, 94, 57, 62, 70],
        'post_score': [5, 43, 23, 23, 51]}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score',
'mid_score', 'post_score'])
print(df)
```

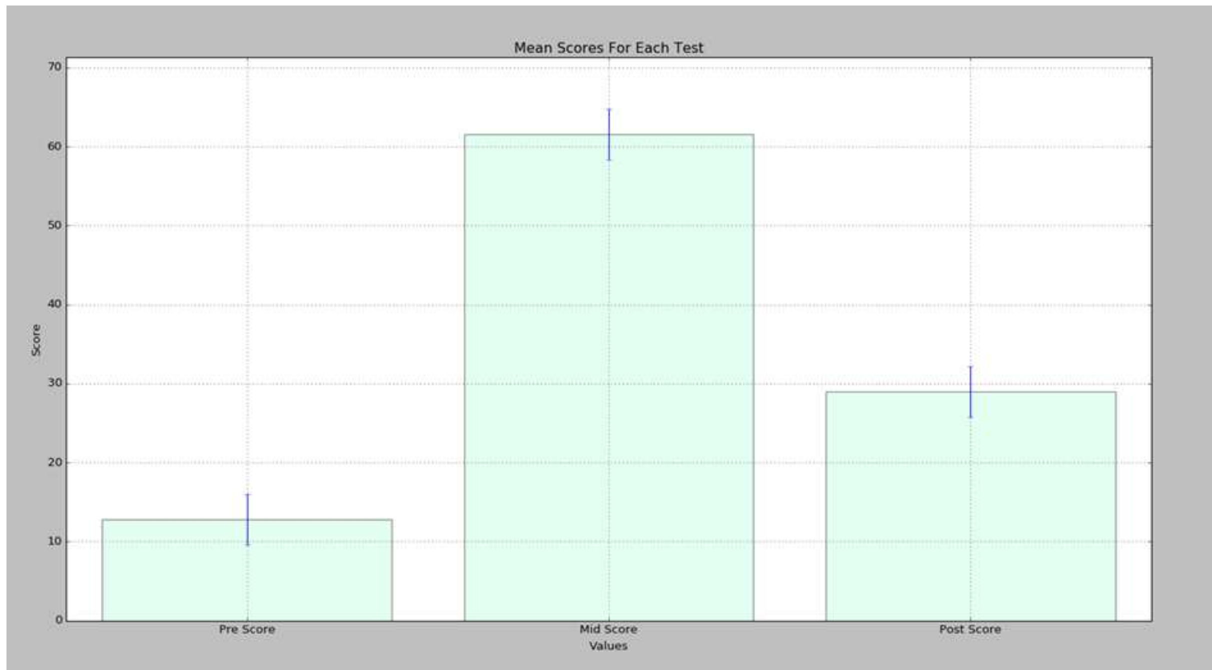
Barres verticales

On souhaite faire le graphique en barres verticale des moyennes sur les trois tests (avec les écart-types)

```
# Création de la liste des moyennes pour chaque variable
mean_values    =    [df['pre_score'].mean(),    df['mid_score'].mean(),
df['post_score'].mean()]
# Création de la liste des écarts pour chaque variable fixé à 0.25 au
dessus de la moyenne
variance  =  [df['pre_score'].mean()  *  0.25,  df['pre_score'].mean()  *
0.25, df['pre_score'].mean() * 0.25]
# Création de la liste des labels des variables
bar_labels = ['Pre Score', 'Mid Score', 'Post Score']
# cadre du graphique
import matplotlib.pyplot as plt
```

```
fig, ax = plt.subplots(figsize=(10,5))   # récupération de l'objet figure et de l'objet axis (subplots renvoie un tuple)
# Création des valeurs l'axe des barres
x_pos = list(range(len(bar_labels)))
# Création du graphique
plt.bar(x_pos,mean_values,yerr=variance,               align='center', color='#C9FFE5',alpha=0.5)
# ajouter des grid (grilles)
plt.grid()
#Graduation et mise en forme de l'axe y
max_y = max(zip(mean_values, variance)) # renvoie un tuple qui prend le maximum de chaque liste. On obttien (3, 5)
plt.ylim([0, (max_y[0] + max_y[1]) * 1.1]) # max_y[0]=3 et max_y[1]=5
# labeliser les axes
plt.ylabel('Score')
plt.xlabel('Values')
plt.xticks(x_pos, bar_labels)
plt.title('Mean Scores For Each Test')
plt.show() # show the graph
plt.savefig("image.png") # Enregistrer le graphique en image dans le répertoire courant.
plt.cla() # Supprime les axes dans la figure courantes
plt.clf() # Supprime tout le graphique actuel avec ses axes mais laisse la fenêtre graphique ouverte.
plt.close()    # Nettoie le graphique actif
fig.clear # supprime la figure.
```

NB : Pour les graphiques successives : utiliser plutôt plt.close()

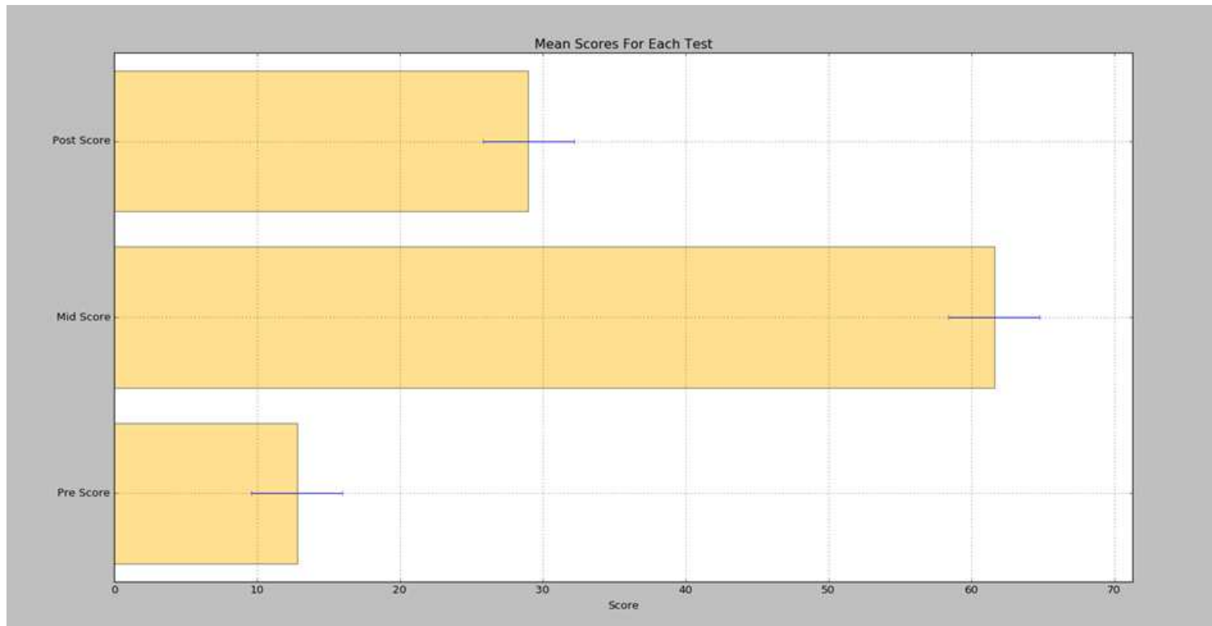


Barres horizontales

En partant du même cadre que le graphique en barres verticales, on a:

```
# Création de la liste des moyennes pour chaque variable
mean_values = [df['pre_score'].mean(), df['mid_score'].mean(),
df['post_score'].mean()]
# Création de la liste des écarts pour chaque variable fixé à 0.25 au
dessus de la moyenne
variance = [df['pre_score'].mean() * 0.25, df['pre_score'].mean() *
0.25, df['pre_score'].mean() * 0.25]
# Création de la liste des labels des variables
bar_labels = ['Pre Score', 'Mid Score', 'Post Score']
# cadre du graphique
# Création de l'axe des barres
y_pos = list(range(len(bar_labels))) # on change x_pos en y_pos (mais
avec même définition)
# Création du graphique
import matplotlib.pyplot as plt
plt.barh(y_pos,mean_values,xerr=variance, align='center',
color='#FFC222',alpha=0.5) # on utilise barh() à la place de bar()
# ajouter des grid (grilles)
plt.grid()
#Graduation de l'axe x
max_x = max(zip(mean_values, variance)) # renvoie un tuple, ici: (3,
5)
plt.xlim([0, (max_x[0] + max_x[1]) * 1.1])
# labeliser les axes
plt.xlabel('Score')
```

```
plt.yticks(y_pos, bar_labels)
plt.title('Mean Scores For Each Test')
plt.show()
plt.savefig("image.png")
plt.cla()
plt.clf()
plt.close()
```



6.1.2. Paramétrage du graphique

6.1.2.1. Description du role de fig, ax= dans un tracé de graphique

fig, ax= : Ce paramétrage permet de référencer à la fois la figure et les axes du graphiques

plt.subplots() : est une function qui renvoie un tuple contenant les objets figures et axes

Ainsi, en utilisant fig, ax = plt.subplots() on décompose le tuple en deux variables fig et ax.

Le paramètre fig est utile dans les cas où l'on veut changer les attributs de la figure ou enregistrer la figure comme l'image (ex : fig.savefig('nomFichier.png').

D'une manière générale, le paramétrage fig, ax = plt.subplots() est plus concis que la formulation fig = plt.figure().

Quant au paramétrage ax, voici un exemple d'utilisation :

ax = fig.add_subplot(111) # Voir signification de 111 ci-dessous (définition des graphiques en cadrans)

Toutefois pour faire des cadrans séparés, la deuxième formule peut être intéressante

6.1.2.2. Liste des couleurs en python

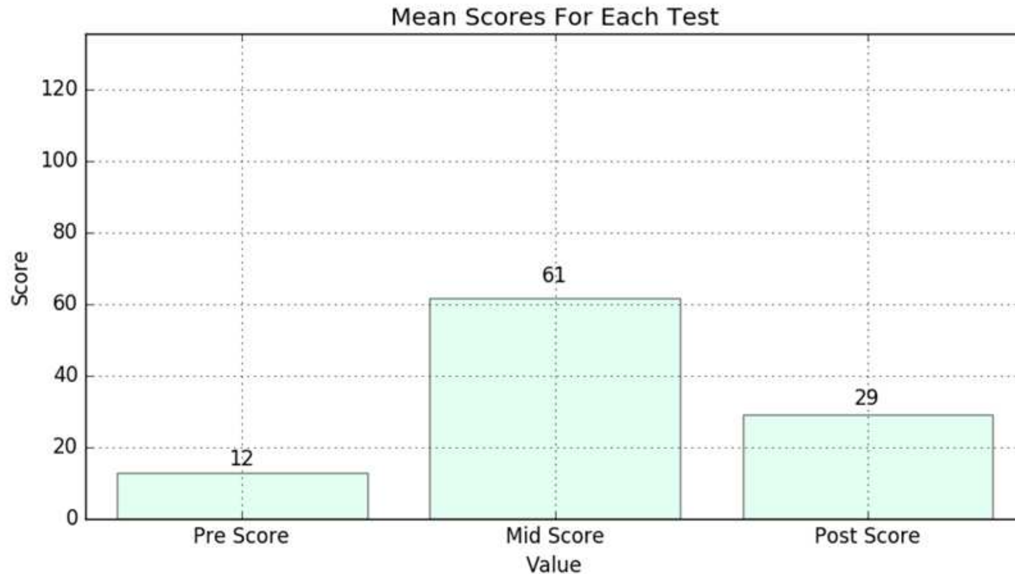
```
import matplotlib
for name, hex in matplotlib.colors.cnames.iteritems():
     print(name, hex)
```

6.1.2.3. Ajouter les valeurs des barres au graphique

Le principe est que l'on récupère chaque valeur de la liste des valeurs et on le place sur le graphique en définissant la position dans le repère.

Ex: soit le graphique suivant:

```
mean_values = [df['pre_score'].mean(), df['mid_score'].mean(),
df['post_score'].mean()]
bar_labels = ['Pre Score', 'Mid Score', 'Post Score']
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(10,5))
width = 0.25
x_pos = list(range(len(bar_labels)))
mybargraph=plt.bar(x_pos,mean_values,align='center',
color='#C9FFE5',alpha=0.5) # le graph est nommé rects1
plt.grid()
max_y = max(zip(mean_values, mean_values))
plt.ylim([0, (max_y[0] + max_y[1]) * 1.1])
plt.ylabel('Score')
plt.xlabel('Value')
plt.xticks(x_pos, bar_labels)
plt.title('Mean Scores For Each Test')
# ajout des valeurs depuis mean_values
# définissons d'abord une petite fonction qui labélise les valeurs
def autolabel(bgraph):
    for bg in bgraph:
        height = bg.get_height()
        ax.text(bg.get_x() + bg.get_width()/2.,
1.05*height,'%d'%int(height), ha='center', va='bottom') # position du
texte sur le graphique
#labelisation:
autolabel(mybargraph) # On met en argument le nom graphique en barres
crée précédemment.
plt.show()
plt.savefig("image.png")
plt.cla()
plt.clf()
plt.close()
```



6.1.2.4. Définition et paramétrage des graphiques en plusieurs cadrans

Pour définir plusieurs cadrans de graphique, on pararmétrise la fonction subplot avec des valeurs qui indiquent à la fois le numéros du subplot, sa position dans le cadran général du graphique. Exemples :

```
import matplotlib.pyplot as plt
plt.subplot(121) # signifie (1X2) pour subplot 1 (1 grid en bas et 2 grid en haut)
plt.subplot(122) # signifie(1X2) pour subplot 2 (1 grid en bas et 2 grid en haut)
```

On distingue plusieurs combainaisons: cadrans symétriques et cadrans non symétriques.

Exemple : cadran symétrique

```
fig = plt.figure()
fig.add_subplot(221)   #top left
fig.add_subplot(222)   #top right
fig.add_subplot(223)   #bottom left
fig.add_subplot(224)   #bottom right
##        # exmple cadran non symérique cas 1:
subplot(2,2,[1 3]) # regroupement subplot 1 et 3
subplot(2,2,2)
subplot(2,2,4)
```

Exemple cadran non symétrique cas 2:

```
subplot(2,2,1:2)# regroupement subplot 1 à 2
subplot(2,2,3)
subplot(2,2,4)
```

Pour mettre en forme la zone du graphique, on utilise la ligne suivante lors de la définition du graphique

```
figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
```

On définit une figure sans numérotation, de taille 8/6, de couleur de fond blanc et de couleur de contour k. On peut remplacre ces couleurs par leur numéro. (Voir application des cadrans multiples plus bas dans le cas de superpositions de graphiques)

6.1.3. Barres de comparaison (de valeurs) par groupe

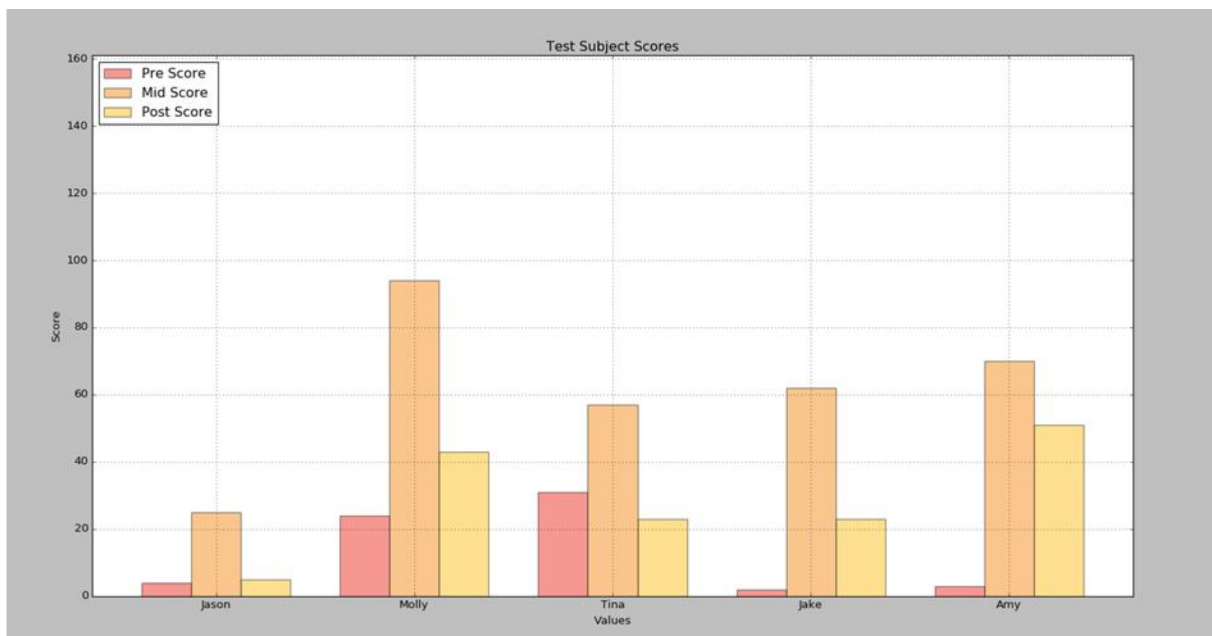
Soit la table de données suivante qui fournit le score de 5 étudiant à trois tests

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'pre_score': [4, 24, 31, 2, 3],
        'mid_score': [25, 94, 57, 62, 70],
        'post_score': [5, 43, 23, 23, 51]}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score',
'mid_score', 'post_score'])
print(df)
```

On souhaite afficher les valeurs de scores pour chaque individu. On a:

```
import matplotlib.pyplot as plt
#Détermination de la position et la taille des barres
pos = list(range(len(df['pre_score']))) # compte le nombre d'élement
dans pre_score et renvoie sous forme de liste
width = 0.25
#Tracé des barres
fig, ax = plt.subplots(figsize=(10,5)) # cadre de base auque les
autres graphiques vont se greffer
#Création des barres  avec pre_score
plt.bar(pos,df['pre_score'],
width,alpha=0.5,color='#EE3224',label=df['first_name'][0])
 #Création des barres  avec mid_score
plt.bar([p + width for p in pos],df['mid_score'],
width,alpha=0.5,color='#F78F1E',label=df['first_name'][1])
 #Création des barres  avec post_score
plt.bar([p + width*2 for p in
pos],df['post_score'],width,alpha=0.5,color='#FFC222',
label=df['first_name'][2])
# Titre de l'axe y
ax.set_ylabel('Score')
ax.set_xlabel('Values')
#Titre du graphique
ax.set_title('Test Subject Scores')
# position des ticks (position des groupes)
ax.set_xticks([p + 1.5 * width for p in pos])
# label de ticks
```

```
ax.set_xticklabels(df['first_name'])
# limite des axes
plt.xlim(min(pos)-width, max(pos)+width*4)
plt.ylim([0,     max(df['pre_score']     +     df['mid_score']     +
df['post_score'])] )
# Définition des légendes
plt.legend(['Pre Score', 'Mid Score', 'Post Score'], loc='upper left')
plt.grid()
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.1.4. Barres de comparaison (de moyennes) par groupe

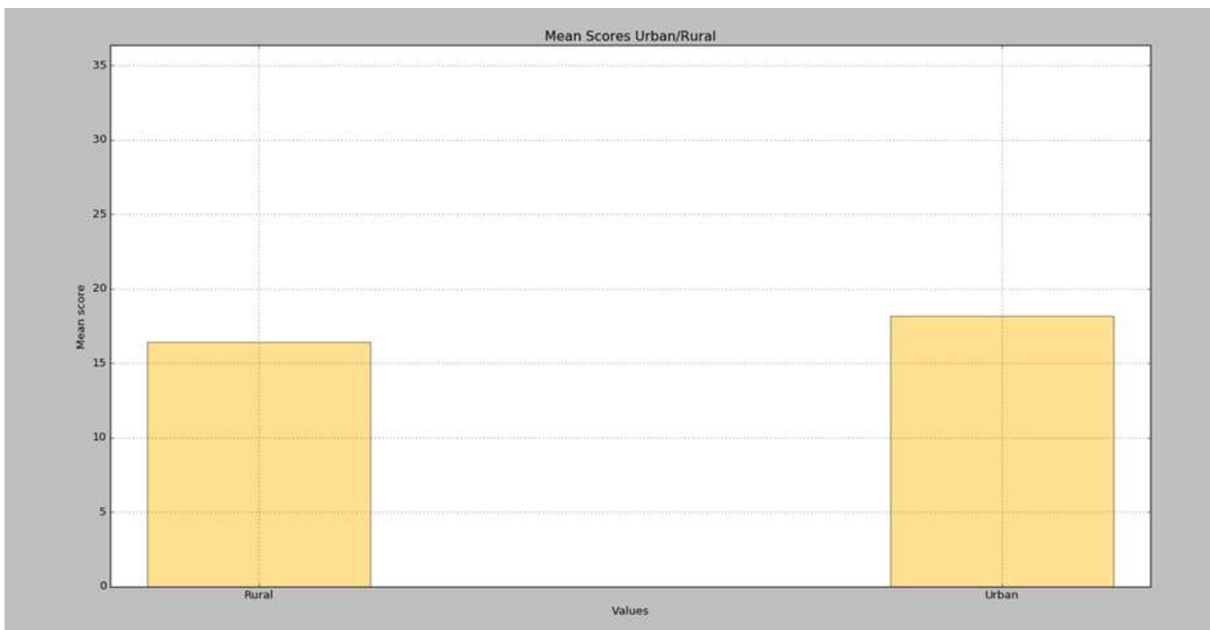
Soit la table de données suivante qui donne les scores de 11 individus

```
raw_data  =  {'first_name':  ['Jason',  'Molly',  'Tina',  'Jake',  'Amy',
'Kader', 'Jean' , 'Philipe', 'Paul', 'Pierre', 'Jacques'],
        'pre_score': [4, 24, 31, 2, 3, 4 , 10 , 25 , 40, 30, 18],
        'mid_score': [25, 94, 57, 62, 70, 12 , 24 , 25 ,64, 28, 20],
        'post_score': [5, 43, 23, 23, 51, 14 , 27 , 26, 70, 80 ,30],
        'Area': ['Rural', 'Urban', 'Rural', 'Urban', 'Urban', 'Rural',
'Urban' , 'Rural', 'Urban', 'Urban', 'Rural']}
df  =  pandas.DataFrame(raw_data,  columns  =  ['first_name',  'pre_score',
'mid_score', 'post_score', 'Area'])
print(df)
```

Nous allons utiliser ce cadre pour examiner plusieurs cas de comparaison.

1- Comparaison de moyenne entre Urban et Rural sur chaque variable 1 à 1 (Rural/Urbain en abcisses)

```
####  graphique pour la variable pre_score
df0=df[['pre_score', 'Area']] # selection de deux variables
df0r=df0[(df0['Area'] == 'Rural')] # sélection Rural
df0r_m=[df0r['pre_score'].mean()] # Calcul de la moyenne rural
df0u=df0[(df0['Area'] == 'Urban')] # sélection urban
df0u_m=[df0u['pre_score'].mean()] # Calcul de la moyenne urban
df0m=df0r_m+df0u_m # liste des deux moyennes
print(df0m)
mlabel=[ 'Rural', 'Urban'] # label des deux moyennes
x_pos = list(range(len(mlabel))) #valeur de l'axe des x
import matplotlib.pyplot as plt
x=plt.bar(x_pos,df0m, 0.3, align='center', color='#FFC222',alpha=0.5)
plt.grid()
max_y = max(zip(df0m, df0m))
plt.ylim([0, (2*max_y[0])])
plt.ylabel('Mean score')
plt.xlabel('Values')
plt.xticks(x_pos, mlabel)
plt.title('Mean Scores Urban/Rural')
plt.show()
plt.cla()
plt.clf()
plt.close()
```



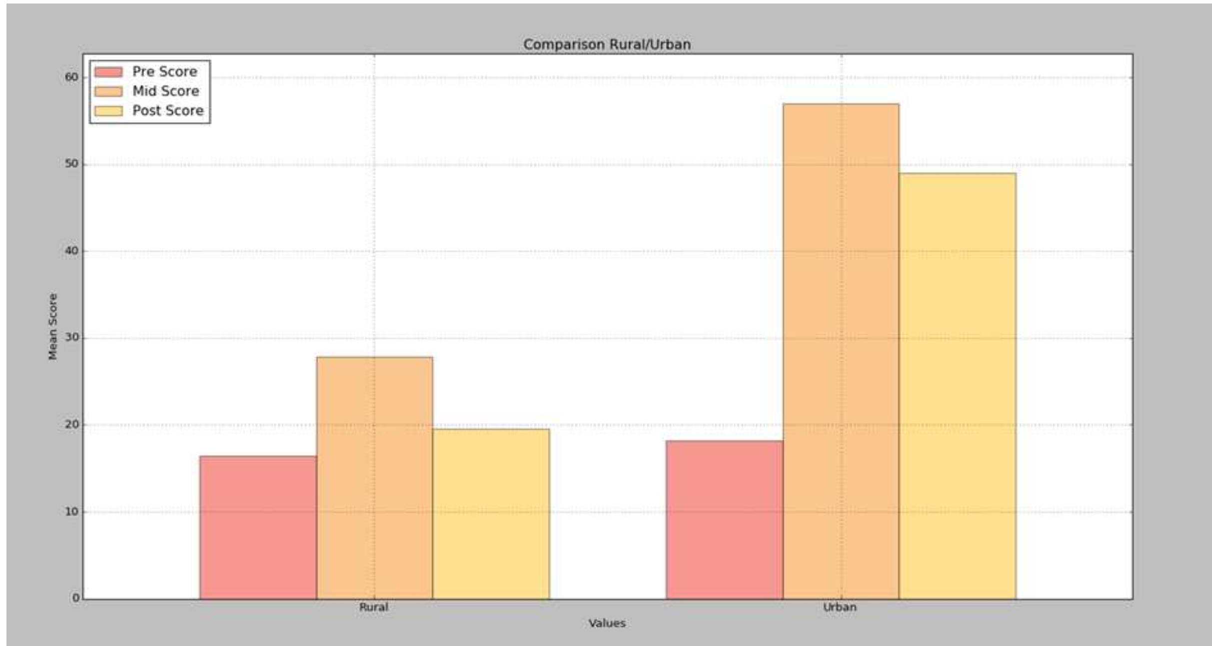
Représentation des trois variables pre_score, mid_score et post_score selon rural et urbain

Pour cela calculons les moyennes Rural/Urbain pour chaque variable

```
###pour la variable pre_score
df0=df[['pre_score', 'Area']] # selection de deux variables
```

```
df0r=df0[(df0['Area'] == 'Rural')] # sélection Rural
df0r_m=[df0r['pre_score'].mean()] # Calcul de la moyenne rural
df0u=df0[(df0['Area'] == 'Urban')] # sélection urban
df0u_m=[df0u['pre_score'].mean()] # Calcul de la moyenne urban
df0m=df0r_m+df0u_m # liste des deux moyennes
#### pour la variable mid_score
df1=df[['mid_score', 'Area']] # selection de deux variables
df1r=df1[(df1['Area'] == 'Rural')] # sélection Rural
df1r_m=[df1r['mid_score'].mean()] # Calcul de la moyenne rural
df1u=df1[(df1['Area'] == 'Urban')] # sélection urban
df1u_m=[df1u['mid_score'].mean()] # Calcul de la moyenne urban
df1m=df1r_m+df1u_m # liste des deux moyennes
#### pour la variable post_score
df2=df[['post_score', 'Area']] # selection de deux variables
df2r=df2[(df2['Area'] == 'Rural')] # sélection Rural
df2r_m=[df2r['post_score'].mean()] # Calcul de la moyenne rural
df2u=df2[(df2['Area'] == 'Urban')] # sélection urban
df2u_m=[df2u['post_score'].mean()] # Calcul de la moyenne urban
df2m=df2r_m+df2u_m # liste des deux moyennes
# préparation du graphique
mlabel=[ 'Rural', 'Urban'] # label des deux moyennes
pos = list(range(len(mlabel))) #valeur de l'axe des x
import matplotlib.pyplot as plt
width = 0.25
# cadre de base du graphique
fig, ax = plt.subplots(figsize=(10,6)) # taille du graphique largeur/hauteur
#Création des barres  avec pre_score
print(df0m, pos)
plt.bar(pos,df0m,  width,alpha=0.5,color='#EE3224',label=mlabel)
 #Création des barres  avec mid_score
plt.bar([p + width for p in pos],df1m, width,alpha=0.5,color='#F78F1E',label=mlabel)
 #Création des barres  avec post_score
plt.bar([p + width*2 for p in pos],df2m,width,alpha=0.5,color='#FFC222', label=mlabel)
# Titre de l'axe y
ax.set_ylabel('Mean Score')
ax.set_xlabel('Values')
#Titre du graphique
ax.set_title('Comparison Rural/Urban')
# position des ticks (position des groupes)
ax.set_xticks([p + 1.5* width for p in pos]) # pour ajuster les ticks
# label de ticks
ax.set_xticklabels(mlabel)
# limite des axes
plt.xlim(min(pos)-width, max(pos)+width*4)
plt.ylim([0, 1.1*max(df0m + df1m + df2m)] )
# Définition des légendes
```

```
plt.legend(['Pre Score', 'Mid Score', 'Post Score'], loc='upper left')
plt.grid()
plt.show()
plt.cla()
plt.clf()
plt.close()
```

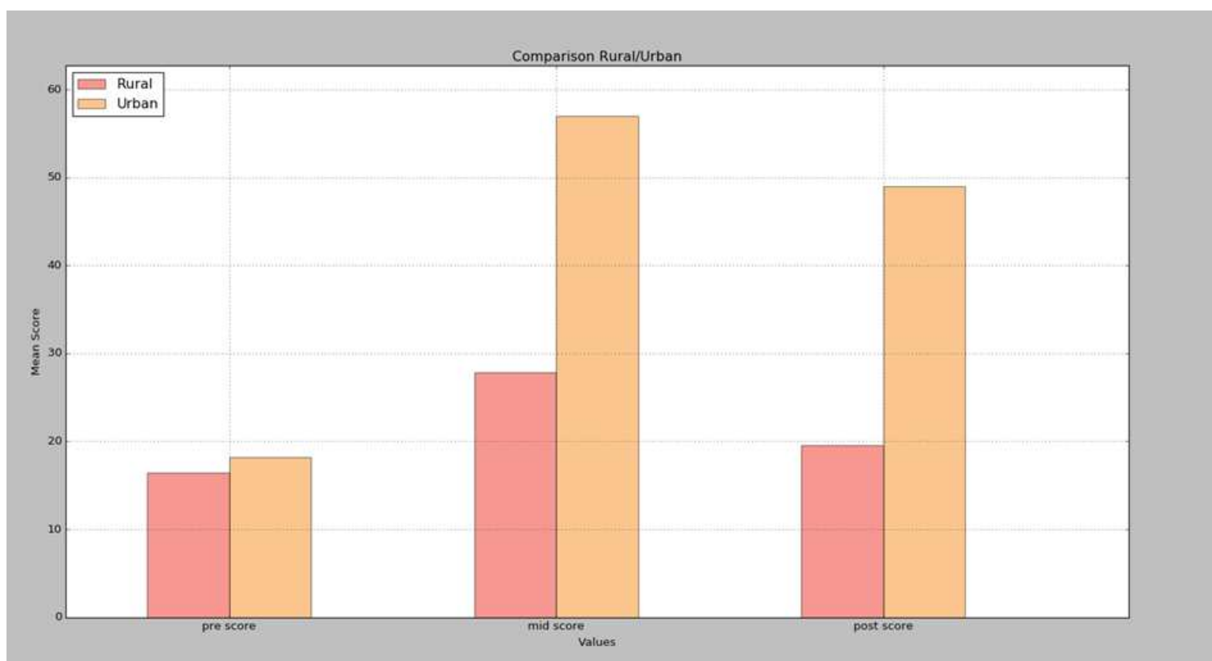


2- Comparaison de la moyenne de chaque variable entre Urban et Rural (les scores sont en abcisses)

Ici, on aura deux subplots: un pour rural et un pour urbain. Le subplot rural est construit à partir d'une liste contenant les moyennes des trois variables calculées sur le milieu rural (de même pour urbain).

```
# construction liste des moyennes rural
dfr=df[(df['Area'] == 'Rural')]
listmr     =     [dfr['pre_score'].mean(),     dfr['mid_score'].mean(),
dfr['post_score'].mean()]
# construction liste des moyennes urbain
dfr=df[(df['Area'] == 'Urban')]
listmu     =     [dfr['pre_score'].mean(),     dfr['mid_score'].mean(),
dfr['post_score'].mean()]
# défintion des labels de l'axe des abscidess
mlabel=['pre score','mid score', 'post score']
# préparation du graphique
pos = list(range(len(mlabel))) #valeur de l'axe des x
import matplotlib.pyplot as plt
width = 0.25
# cadre de base du graphique
```

```
fig, ax = plt.subplots(figsize=(10,6)) # taille du graphique largeur/hauteur
#Création des barres  avec rural
plt.bar(pos,listmr,  width,alpha=0.5,color='#EE3224',label=mlabel)
 #Création des barres  avec urban
plt.bar([p + width for p in pos],listmu, width,alpha=0.5,color='#F78F1E',label=mlabel)
# Titre de l'axe y
ax.set_ylabel('Mean Score')
ax.set_xlabel('Values')
#Titre du graphique
ax.set_title('Comparison Rural/Urban')
# position des ticks (position des groupes)
ax.set_xticks([p + 1.* width for p in pos])
# label de ticks
ax.set_xticklabels(mlabel)
# limite des axes
plt.xlim(min(pos)-width, max(pos)+width*4)
plt.ylim([0, 1.1*max(listmr + listmu)] )
# Définition des légendes
plt.legend(['Rural', 'Urban'], loc='upper left')
plt.grid()
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.1.5. Barres inversées (modèle pyramides des âges)

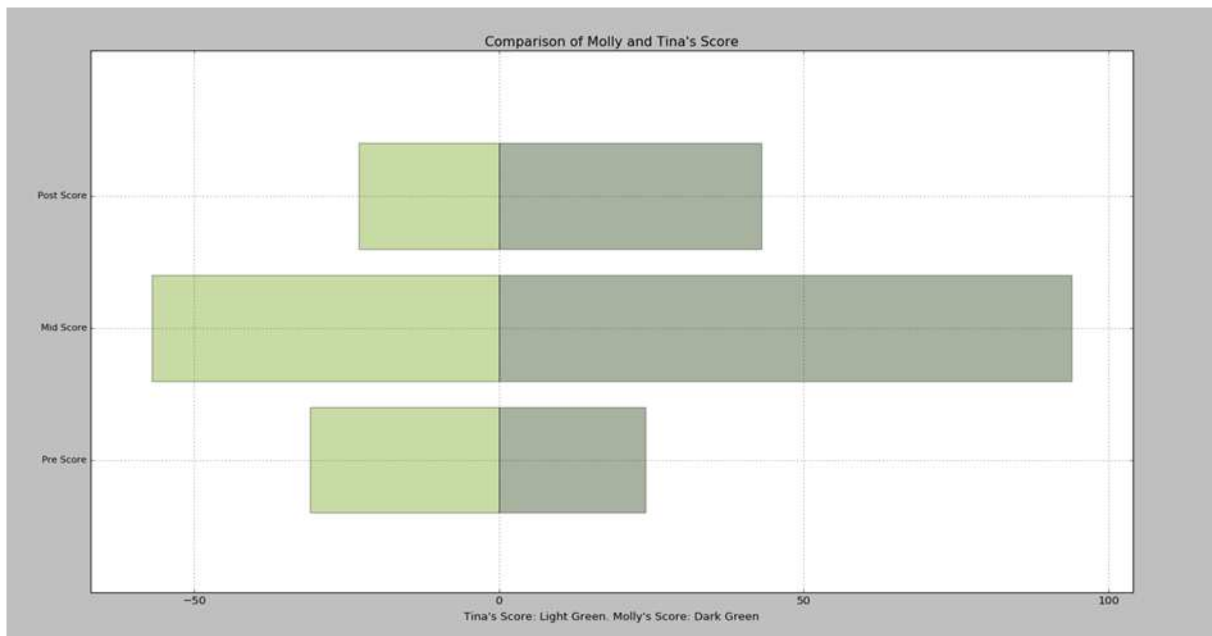
Soit la table de données suivante qui fournit le score de 5 étudiants à trois tests

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'pre_score': [4, 24, 31, 2, 3],
        'mid_score': [25, 94, 57, 62, 70],
        'post_score': [5, 43, 23, 23, 51]}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score',
'mid_score', 'post_score'])
print(df)
```

Faisons un graphique de barre horizontale comparant les scores entre Molly et Tina.

Le graphique se présente comme la pyramide des âges. On a les étapes suivantes:

```
#- Récupération des données sur Molly et Tina ( 2ième et 3ième ligne)
x1 = df.ix[1, 1:]
x2 = df.ix[2, 1:]
# Création des labels pour les trois types de scores
bar_labels = ['Pre Score', 'Mid Score', 'Post Score']
# Création de l'objet graphique
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(8,6))
# Réglage, graduation et mise en forme de l'axe des barres (axe y),
y_pos = numpy.arange(len(x1))
y_pos = [x for x in y_pos]
plt.yticks(y_pos, bar_labels, fontsize=10)
# Création des barres horizontales pour x1 ( centered with alpha 0.4
and color green)
plt.barh(y_pos,  x1, align='center', alpha=0.4,   color='#263F13')
# Création des barres horizontales pour x2(   negative centered with
alpha 0.4  and color green)
plt.barh(y_pos,  -x2, align='center', alpha=0.4,   color='#77A61D')
# Labellisation et mise en forme du graphique
plt.xlabel('Tina\'s Score: Light Green. Molly\'s Score: Dark Green')
t = plt.title('Comparison of Molly and Tina\'s Score')
plt.ylim([-1,len(x1)+0.1])
plt.xlim([-max(x2)-10, max(x1)+10])
plt.grid()
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.1.6. Superposition de plusieurs graphiques en barres dans une fenêtre graphique unique

Pour faire une combainaison des graphiques, on utilite la fonction subplots. Mais deux principaux cas se présentent: le cas où les deux graphiques apparaissent dans un même cadran et le cas où les graphiques apparaissent sur des cadrans séparés.

Ces deux types de combinaison de graphique peuvet se faire selon qu'il s'agit d'un graphique en barre, histogramme, diagramme circulaire, nuage de points et ligne.

Dans cette section, nous présentons le cas de graphiques en barres.

Exemple

Soit la table de données suivante qui donne les scores des étudiants à trois tests : pre_score mid score et post score

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy',
'Kader', 'Jean' , 'Philipe', 'Paul', 'Pierre', 'Jacques'],
        'pre_score': [4, 24, 31, 2, 3, 4 , 10 , 25 , 40, 30, 18],
        'mid_score': [25, 94, 57, 62, 70, 12 , 24 , 25 ,64, 28, 20],
        'post_score': [5, 43, 23, 23, 51, 14 , 27 , 26, 70, 80 ,30],
        'Area': ['Rural', 'Urban', 'Rural', 'Urban', 'Urban', 'Rural',
'Urban' , 'Rural', 'Urban', 'Urban', 'Rural']}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score',
'mid_score', 'post_score', 'Area'])
print(df)
```

On souhaite comparer les moyenne des trois tests par zone (à l'intérieur de chaque zone). On a alors deux sous-graphiques: le sous-graphique pour la zone rurale et le sous-graphique pour la zone urbaine.

On peut soit décider de représenter les deux sous-graphiques dans le même cadran ou bien dans deux cadrans différents

Dans le premier cas , on peut faire une comparaison à l'intérieur des zones (rural/urbain) mais aussi une comparaison entre les valeurs moyennes à l'intérieur de chaque zone alors que dans le second cas, on compare uniquement les valeurs moyennes à l'intérieur de chaque zone

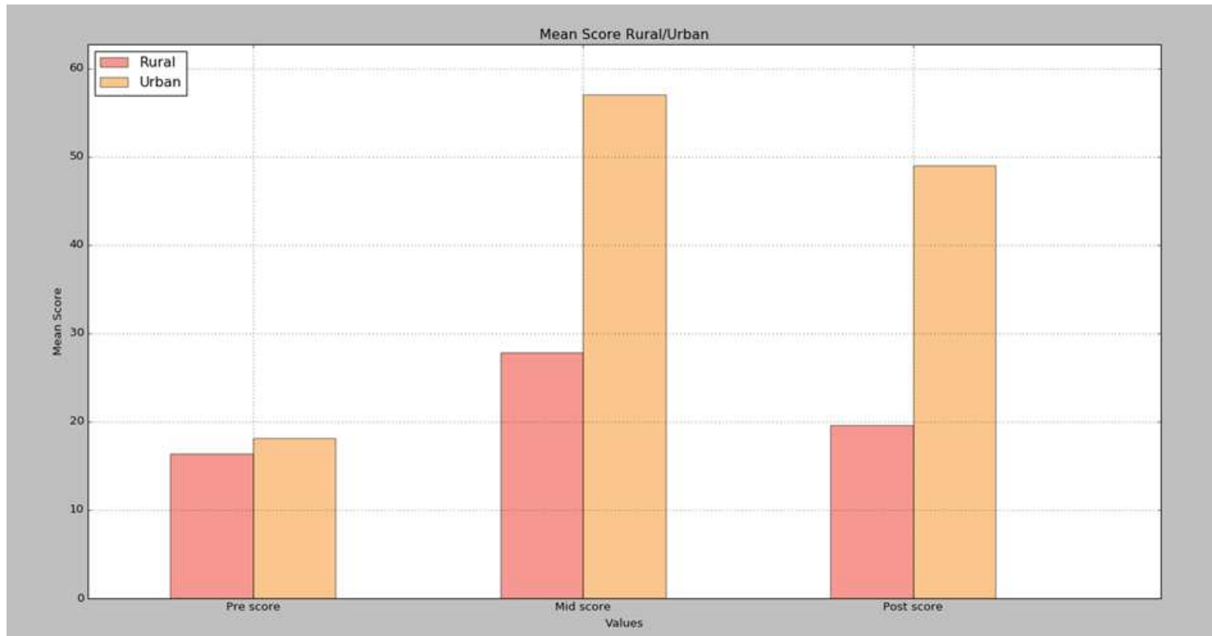
Appliquons ces deux cas en comparant rural et urbain.

```
# Préparation des données:
# liste des moyennes rural
listmr = [df['pre_score'][df['Area'] == 'Rural'].mean(), df['mid_score'][df['Area'] == 'Rural'].mean(), df['post_score'][df['Area'] == 'Rural'].mean()]
# liste des moyennes urbain
listmu = [df['pre_score'][df['Area'] == 'Urban'].mean(), df['mid_score'][df['Area'] == 'Urban'].mean(), df['post_score'][df['Area'] == 'Urban'].mean()]
# défintion des labels de l'axe des abscidess
mlabel=['Pre score','Mid score', 'Post score']
# préparation du graphique
pos = list(range(len(mlabel))) #valeur de l'axe des x
import matplotlib.pyplot as plt
width = 0.25
```

6.1.6.1. Tracer des deux graphiques en barres dans le même cadran

```
#Définition du cadran de base du graphique
fig, ax = plt.subplots(figsize=(20,10)) # taille du graphique largeur/hauteur
plt.bar(pos,listmr, width,alpha=0.5,color='#EE3224',label=mlabel) #Création des barres avec rural
plt.bar([p + width for p in pos],listmu, width,alpha=0.5,color='#F78F1E',label=mlabel) #Création des barres avec urban
ax.set_ylabel('Mean Score') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Mean Score Rural/Urban') #Titre du graphique
ax.set_xticks([p + 1.* width for p in pos]) # position des ticks (position des groupes)
ax.set_xticklabels(mlabel) # label de ticks
plt.xlim(min(pos)-width, max(pos)+width*4) # limite des axes
plt.ylim([0, 1.1*max(listmr + listmu)] )
plt.legend(['Rural', 'Urban'], loc='upper left') # Définition des légendes
```

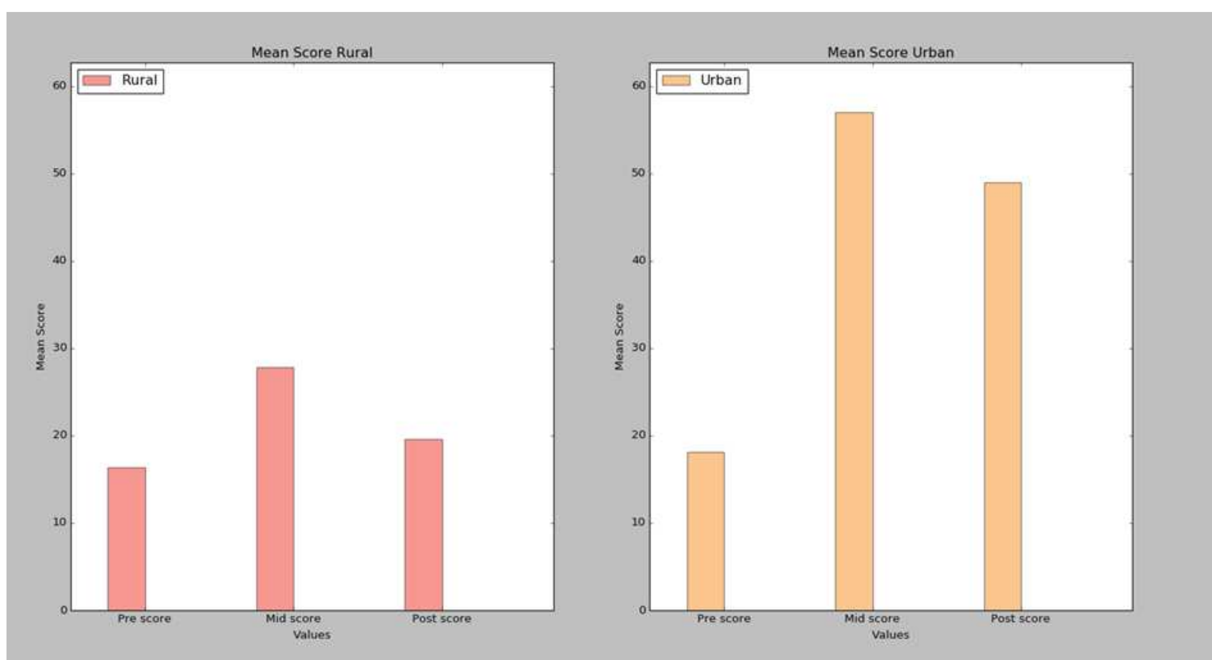
```
plt.grid()
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.1.6. 2. Tracer des deux graphiques en barres dans deux cadrans séparés

```
fig =plt.figure('myfigure',figsize=(20,10)) # Nom et taille de la figure largeur/hauteur
# Cadran rural
ax = fig.add_subplot(121) # position du subplot dans le cadran (1X2) pour subplot 1 (1 grid en bas et 2 grid en haut)
plt.bar(pos,listmr,     width,alpha=0.5,color='#EE3224',label=mlabel) #Création des barres  avec rural
ax.set_ylabel('Mean Score') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Mean Score Rural') #Titre du cadran
ax.set_xticks([p + 1.* width for p in pos]) # position des ticks (position des groupes)
ax.set_xticklabels(mlabel) # label de ticks
plt.xlim(min(pos)-width, max(pos)+width*4) # limite des axes
plt.ylim([0, 1.1*max(listmr + listmu)] )
plt.legend(['Rural'], loc='upper left') # Définition des légendes
#Cadran urbain
ax = fig.add_subplot(122) # position du subplot dans le cadran (1X2) pour subplot 2 (1 grid en bas et 2 grid en haut)
plt.bar(pos,listmu,     width,alpha=0.5,color='#F78F1E',label=mlabel) #Création des barres  avec urban
ax.set_ylabel('Mean Score') # Titre de l'axe y
```

```
ax.set_xlabel('Values')
ax.set_title('Mean Score Urban') #Titre du cadran
ax.set_xticks([p + 1.* width for p in pos]) # position des ticks
(position des groupes)
ax.set_xticklabels(mlabel) # label de ticks
plt.xlim(min(pos)-width, max(pos)+width*4) # limite des axes
plt.ylim([0, 1.1*max(listmr + listmu)] )
plt.legend(['Urban'], loc='upper left') # Définition des légendes

plt.savefig("myfigure.png") # Enregistrer le graphique en image
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.2. Les Histogrammes

6.2.1. Histogrammes simples et histogrammes combinés

Soit la table de données suivante.

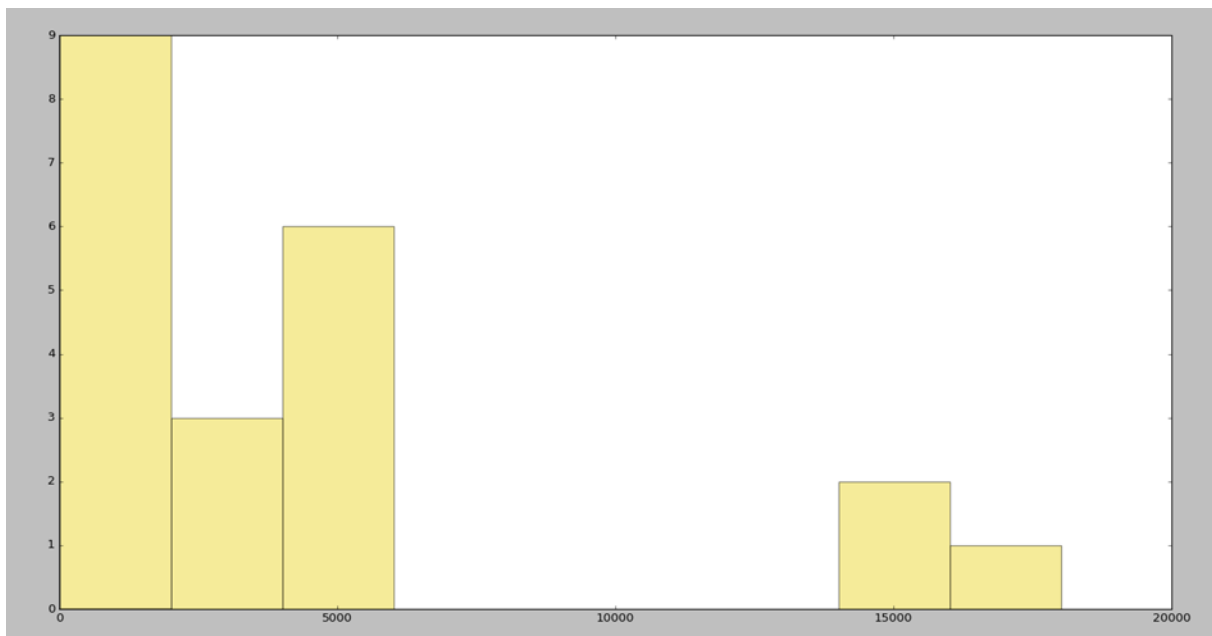
```
df = pandas.read_csv('5kings_battles_v1.csv')
print(df.head())
```

On veut tracer l'histogramme de deux variables attacker_size et defender_size.

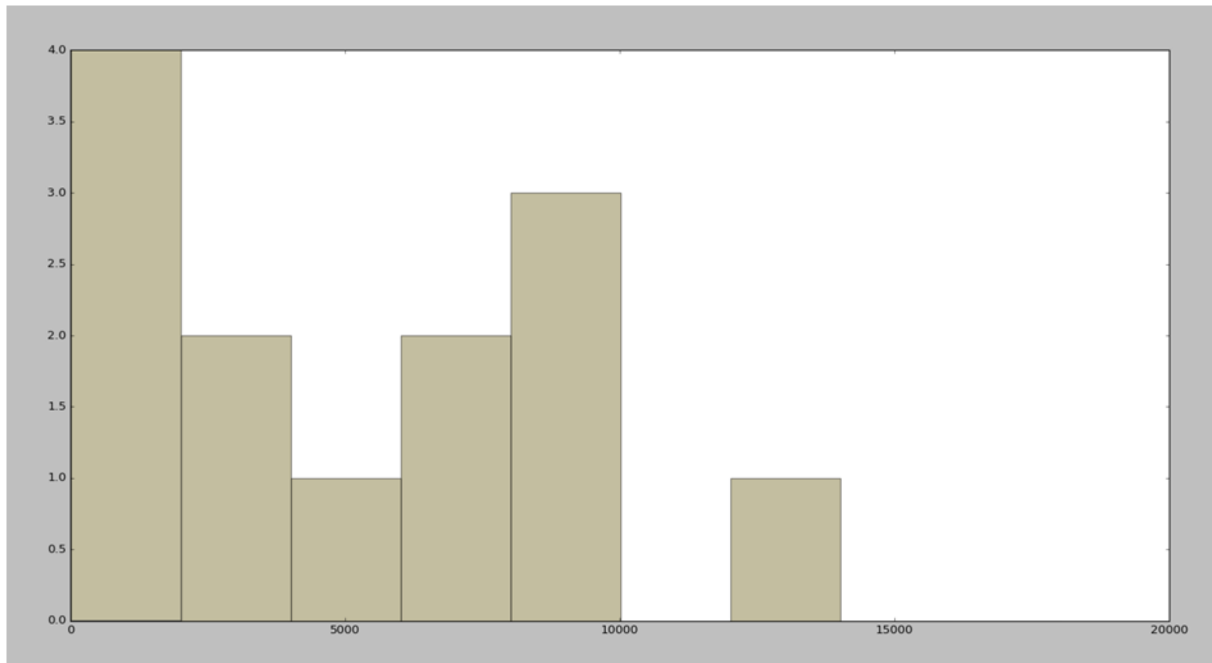
Pour cela, récupérons ces deux variables dans deux sous-tables respectivement (Nb: on garde uniquement les observations pour lesquelles attacker_size est inférieur à 90000)

```
data1 = df['attacker_size'][df['attacker_size'] < 90000]
```

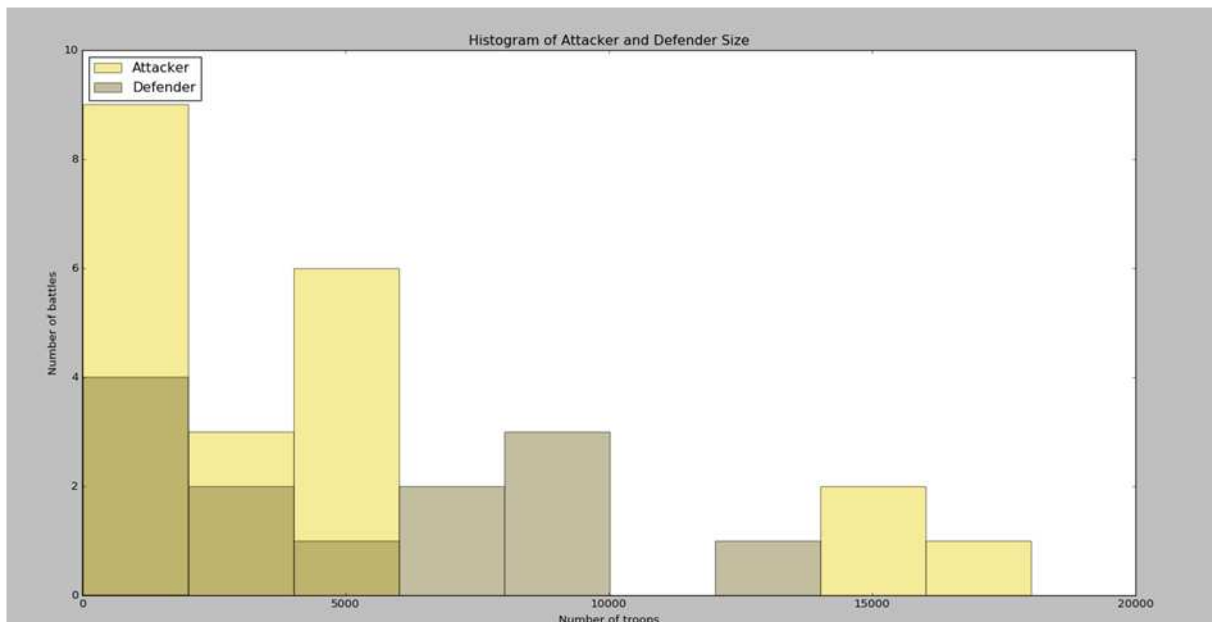
```
data2 = df['defender_size'][df['attacker_size'] < 90000]

# Ensuite essayons de définir un nombre de bin compris entre le min de attacker_size et le max de defender_size
nbins = numpy.arange(data1.min(), data2.max(), 2000) # l'écart entre les bins est de 2000 (histogramme avec avec bins de taille fixe)
#nbins =5
# On peut aussi choisir un nombre fixe de bins comme suit :
#nbins = numpy.linspace(min(data1 + data2), max(data1 + data2),10) # Nombre de bins fixé à 10.
#### Tracer des histogrammes individuels
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import math
fig, ax = plt.subplots(figsize=(20,10)) # taille du graphique largeur/hauteur
# Histogramme pour attacker size
plt.hist(data1, bins=nbins, alpha=0.5,color='#EDD834',label='Attacker')
plt.show()
```




```
# Histogramme pour defender size
plt.hist(data2, bins=nbins,alpha=0.5,color='#887E43',label='Defender')
plt.show()
```




```
#### Tracer des histogrammes combinés
plt.ylim([0, 10]) # pour ajuster la hauteur aux valeurs des féquences
( à modifier au besoin)
plt.title('Histogram of Attacker and Defender Size')
plt.xlabel('Number of troops')
plt.ylabel('Number of battles')
plt.legend(['Attacker','Defender'], loc='upper left')
#plt.legend(loc='upper right')# Le label de l'hist tien lieu du legend
si aucune valeur n'est indiqué
plt.show() # Pour afficher les deux histogrammes, il faut désactiver
l'affichage dans les cas indivuduels. Désactiver simplement plt.show()
dans chaque cas
plt.cla()
plt.clf()
plt.close()
```



6.2.2. Combinaison de plusieurs histogrammes dans un seul cadre graphique

Soit la table de données suivante

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy', 'Kader', 'Jean' , 'Philipe', 'Paul', 'Pierre', 'Jacques'],
        'pre_score': [4, 24, 31, 2, 3, 4 , 10 , 25 , 40, 30, 18],
        'mid_score': [25, 94, 57, 62, 70, 12 , 24 , 25 ,64, 28, 20],
        'post_score': [5, 43, 23, 23, 51, 14 , 27 , 26, 70, 80 ,30],
        'Area': ['Rural', 'Urban', 'Rural', 'Urban', 'Urban', 'Rural', 'Urban' , 'Rural', 'Urban', 'Urban', 'Rural']}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score', 'mid_score', 'post_score', 'Area'])
print(df)
```

Dans un premier temps, on veut faire l'histogramme des trois variables pre_score, mid_score et post_score dans le même cadran et dans trois cadran différents

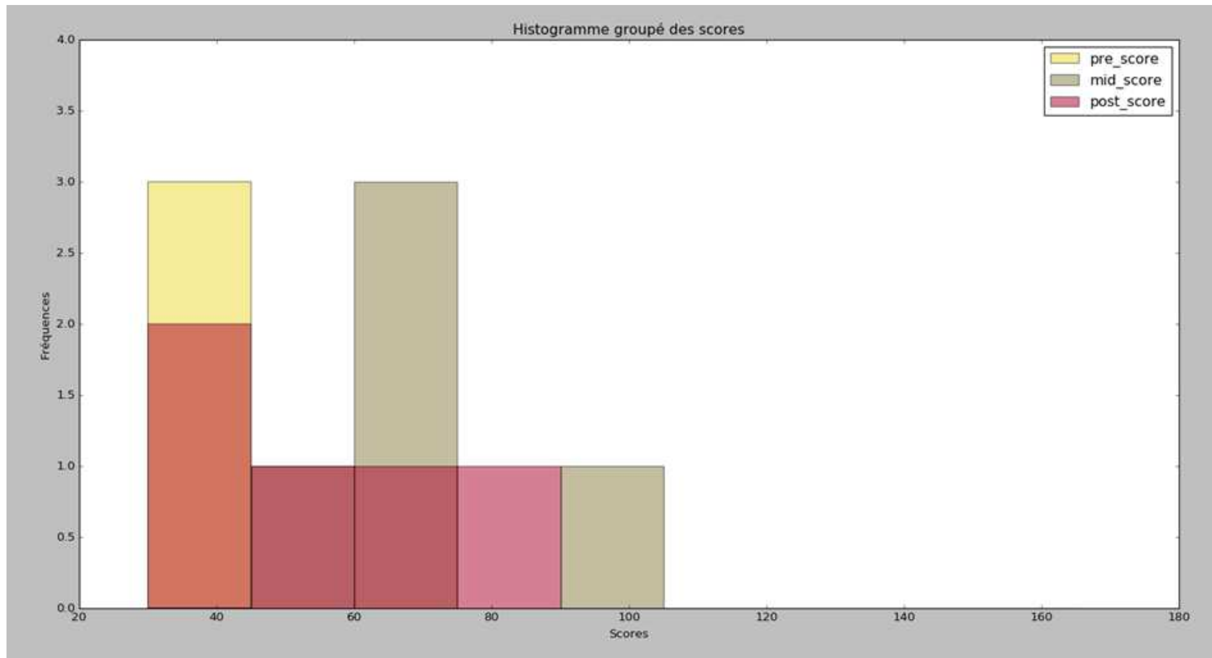
Dans un second temps, on souhaite faire l'histogramme de la variable pre_score sur le milieu rural et urbain dans le même cadran et dans deux cadrans séparés

Dans un troisième temps, on veut faire l'histogramme des trois variables pre_score, mid_score et post_score sur le milieu rural et urbain dans le même cadran et dans deux cadrans séparés (un cadre pour rural et un cadre pour urbain)

Dans un quatrième temps, on veut faire l'histogramme de chaque variable sur le milieu rural et urbain dans 6 cadrans différents

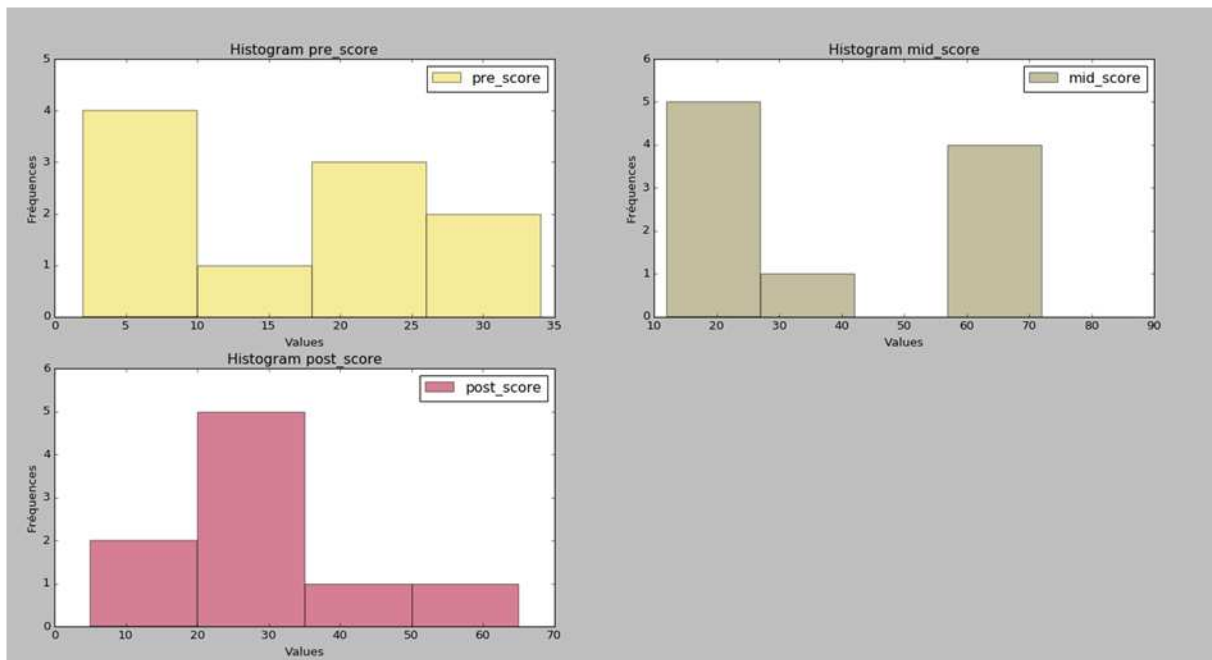
Premier cas: histogramme des trois variables pre_score, mid_score et post_score dans le même cadran et dans trois cadran différents

```
import numpy as np
import matplotlib.pyplot as plt
Pour cela, récupérons ces trois variables dans trois sous-tables
respectivement
data1 = df['pre_score']
data2 = df['mid_score']
data3 = df['post_score']
# Définition du nombre de bins
nbins = np.arange(min(data1+data2+data3), max(data1+data2+data3), 15)
# bin d'écart fixe 15 ( défini sur l'ensemble des valeurs des trois
scores
#nbins = np.linspace(min(data1 + data2+data3), max(data1 +
data2+data3),5) # Nombre de bins fixé à 5.
#nbins =5
#### Tracé des histogrammes
############## histogramme dans le même cadran

fig, ax = plt.subplots(figsize=(20,10)) # taille du graphique
largeur/hauteur
plt.hist(data1, bins=nbins,
alpha=0.5,color='#EDD834',label='pre_score')
plt.hist(data2, bins=nbins,alpha=0.5,color=
'#887E43',label='mid_score')
plt.hist(data3, bins=nbins,alpha=0.5,color=
'#AF002A',label='post_score')
plt.ylim([0, 4]) # pour ajuster la hauteur aux valeurs des féquences (
à modifier au besoin)
plt.title('Histogramme groupé des scores')
plt.xlabel('Scores')
plt.ylabel('Fréquences')
# plt.legend(['pre_score2','mid_score','post_score'], loc='upper
left')
plt.legend(loc='upper right') # Le label de l'hist tien lieu du legend
si aucune valeur n'est indiquée
plt.show()
plt.cla()
plt.clf()
plt.close()
```




```
############## histogramme dans des cadrans séparés
fig =plt.figure('myfigure',figsize=(20,10)) # Nom et taille de la figure largeur/hauteur
# Cadran pre_score
ax = fig.add_subplot(221)
nbins = np.arange(min(data1), max(data1), 8)
#nbins =5
plt.hist(data1, bins=nbins, alpha=0.5,color='#EDD834',label='pre_score')
plt.ylim([0, 5] )
ax.set_ylabel('Fréquences') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Histogram pre_score') #Titre du cadran
plt.legend(loc='upper right') # à enlever si pas nécessaire
# Cadran mid_score
ax = fig.add_subplot(222)
nbins = np.arange(min(data2), max(data2), 15)
plt.hist(data2, bins=nbins, alpha=0.5,color='#887E43',label='mid_score')
plt.ylim([0, 6] )
ax.set_ylabel('Fréquences') # Titre de l'axe y
ax.set_title('Histogram mid_score') #Titre du cadran
ax.set_xlabel('Values')
plt.legend(loc='upper right') # à enlever si pas nécessaire
# Cadran mid_score
ax = fig.add_subplot(223)
nbins = np.arange(min(data3), max(data3), 15)
plt.hist(data3, bins=nbins,alpha=0.5,color= '#AF002A',label='post_score')
plt.ylim([0, 6] )
```

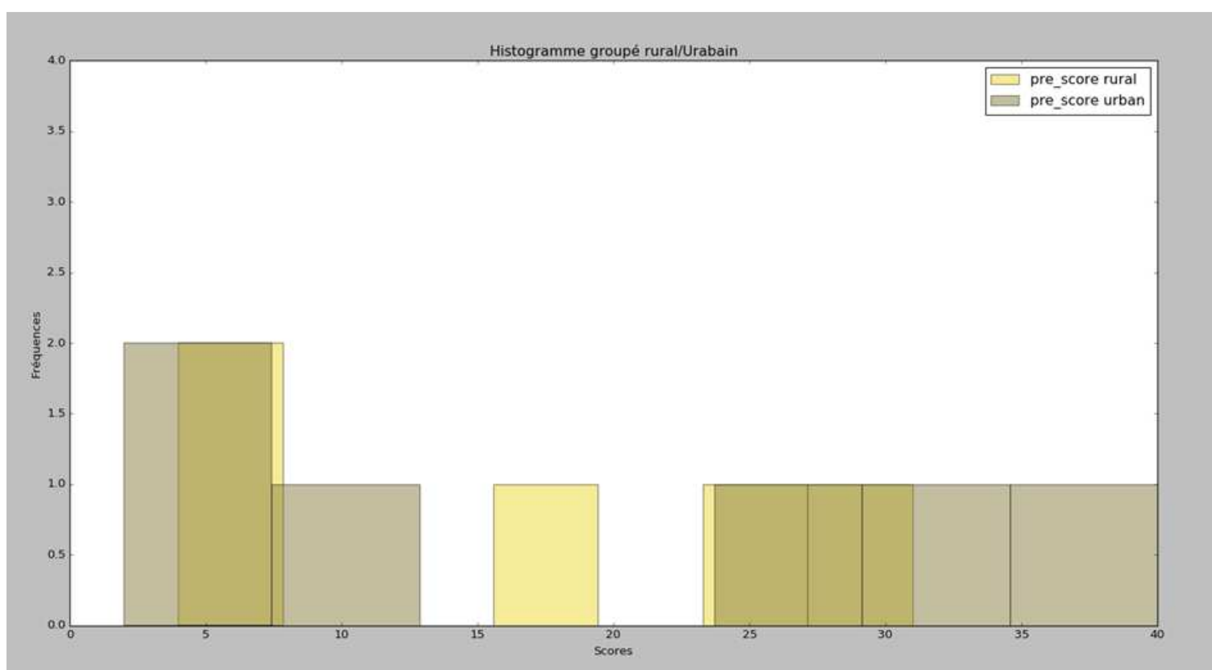
```
ax.set_ylabel('Fréquences') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Histogram post_score') #Titre du cadran
plt.legend(loc='upper right') # à enlever si pas nécessaire
plt.show()
plt.cla()
plt.clf()
plt.close()
```



Deuxième cas : histogramme de la variable pre_score sur le milieu rural et urbain dans le même cadran et dans deux cadrans séparés

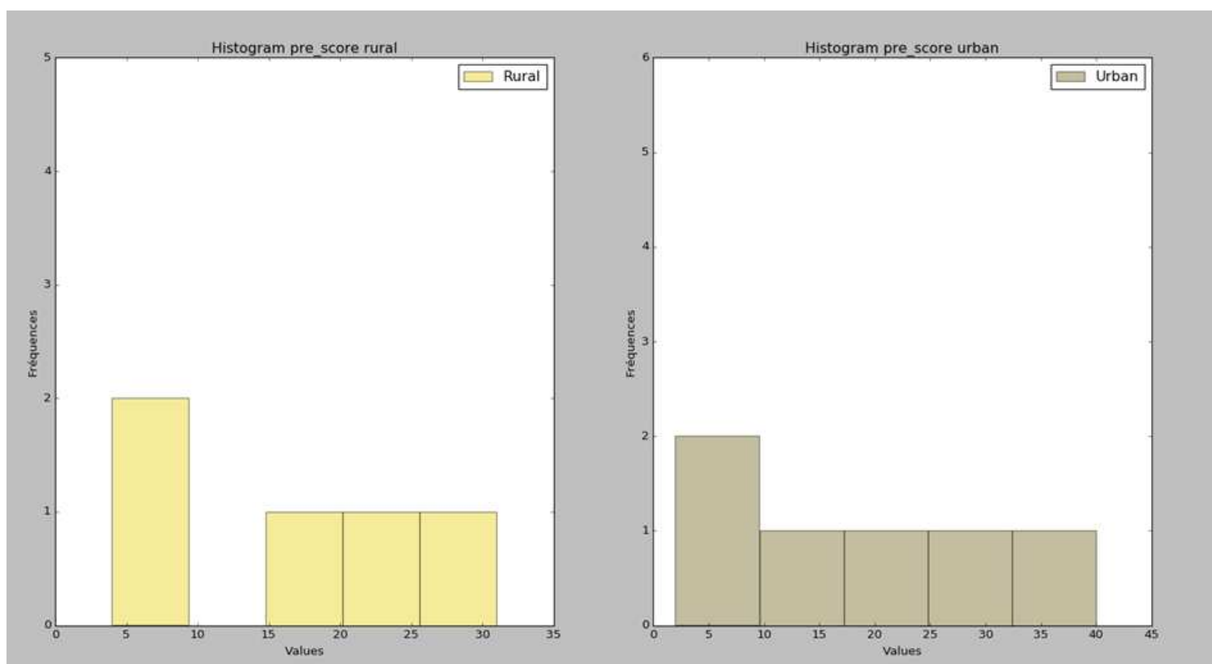
```
import numpy as np
import matplotlib.pyplot as plt
Pour cela, récupérons les valeurs de pre_score pour rural et urbain
data1 = df['pre_score'][df['Area'] =='Rural']
data2 = df['pre_score'][df['Area'] =='Urban']
# Définition du nombre de bins
# nbins = np.arange(min(data1+data2), max(data1+data2), 15) # bin d'écart fixe 15 ( défini sur l'ensemble des valeurs des trois scores
#nbins = np.linspace(min(data1 + data2), max(data1 + data2),15) # Nombre de bins fixé à 5.
nbins =7
#### Tracer des histogrammes
########### histogramme dans le même cadran

fig, ax = plt.subplots(figsize=(10,6)) # taille du graphique largeur/hauteur
```

```
plt.hist(data1,                                          bins=nbins,
alpha=0.5,color='#EDD834',label='pre_score rural')
plt.hist(data2,                              bins=nbins,alpha=0.5,color=
'#887E43',label='pre_score urban')
plt.ylim([0, 4]) # pour ajuster la hauteur aux valeurs des féquences (
à modifier au besoin)
plt.title('Histogramme groupé rural/Urabain')
plt.xlabel('Scores')
plt.ylabel('Fréquences')
plt.legend(loc='upper right')
plt.show()
plt.cla()
plt.clf()
plt.close()
```




```
############## histogramme dans des cadrans séparés
fig  =plt.figure('myfigure',figsize=(10,6))  #  Nom  et  taille  de  la
figure largeur/hauteur
# Cadran rural
ax = fig.add_subplot(121)
#nbins = np.arange(min(data1), max(data1), 8)
nbins =5
plt.hist(data1,  bins=nbins, alpha=0.5,color='#EDD834',label='Rural')
plt.ylim([0, 5] )
ax.set_ylabel('Fréquences') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Histogram pre_score rural') #Titre du cadran
plt.legend(loc='upper right') # à enlever si pas nécessaire
# Cadran urbain
```

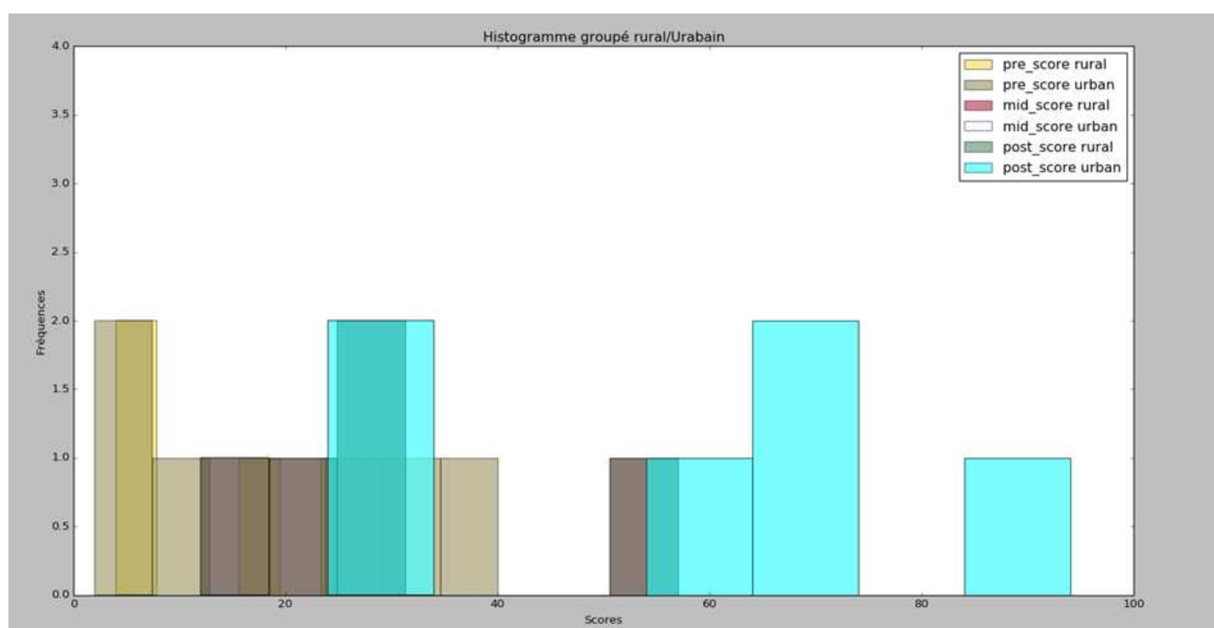
```
ax = fig.add_subplot(122)
#nbins = np.arange(min(data2), max(data2), 15)
nbins =5
plt.hist(data2,  bins=nbins, alpha=0.5,color='#887E43',label='Urban')
plt.ylim([0, 6] )
ax.set_ylabel('Fréquences') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Histogram pre_score urban') #Titre du cadran
plt.legend(loc='upper right') # à enlever si pas nécessaire
plt.show()
plt.cla()
plt.clf()
plt.close()
```



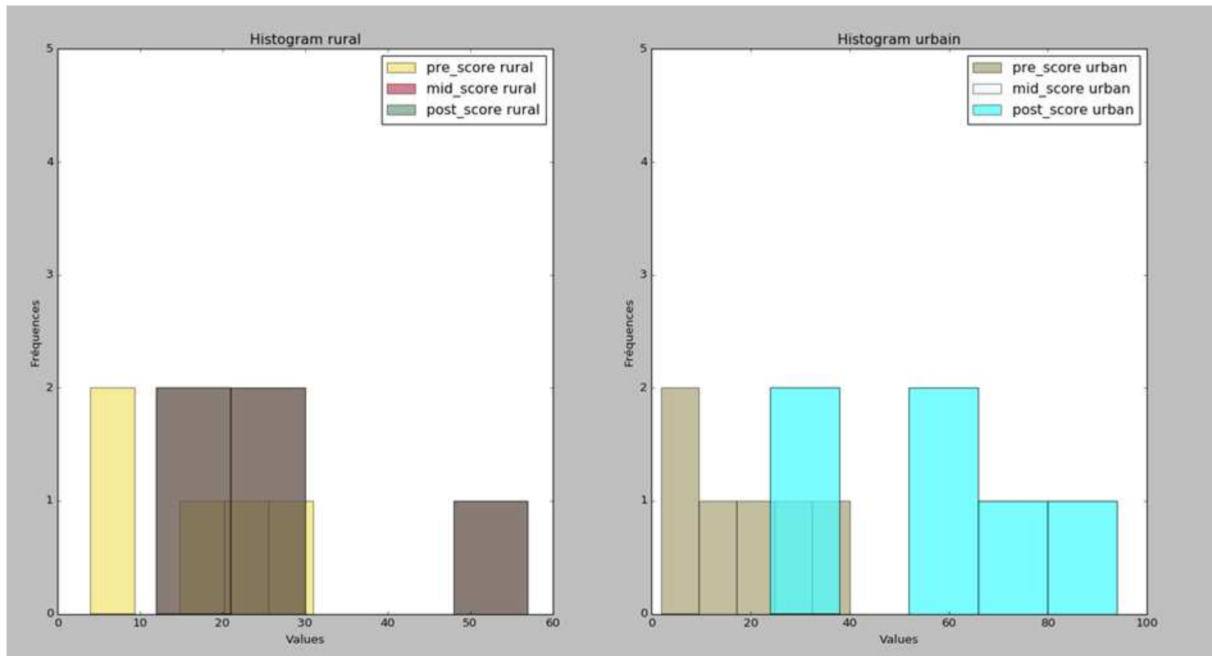
Troisième cas: histogramme des trois variables pre_score, mid_score et post_score sur le milieu rural et urbain dans le même cadran et dans deux cadrans séparés (un cadre pour rural et un cadre pour urbain)

```
import numpy as np
import matplotlib.pyplot as plt
Pour cela, récupérons les valeurs de pre_score, mid_score et post_score pour rural et urbain
data1pre = df['pre_score'][df['Area'] =='Rural']
data2pre = df['pre_score'][df['Area'] =='Urban']
data1mid = df['mid_score'][df['Area'] =='Rural']
data2mid = df['mid_score'][df['Area'] =='Urban']
data1post = df['mid_score'][df['Area'] =='Rural']
data2post = df['mid_score'][df['Area'] =='Urban']
# Définition du nombre de bins
```

```
# nbins = np.arange(min(data1+data2), max(data1+data2), 15) # bin
d'écart fixe 15 ( défini sur l'ensemble des valeurs des trois scores
#nbins = np.linspace(min(data1 + data2), max(data1 + data2),15) #
Nombre de bins fixé à 5.
nbins =7
#### Tracer des histogrammes
############## histogramme dans le même cadran
fig, ax = plt.subplots(figsize=(10,6)) # taille du graphique
largeur/hauteur
plt.hist(data1pre, bins=nbins,
alpha=0.5,color='#EDD834',label='pre_score rural')
plt.hist(data2pre, bins=nbins,alpha=0.5,color=
'#887E43',label='pre_score urban')
plt.hist(data1mid, bins=nbins,
alpha=0.5,color='#AF002A',label='mid_score rural')
plt.hist(data2mid, bins=nbins,alpha=0.5,color=
'#F0F8FF',label='mid_score urban')
plt.hist(data1post, bins=nbins,
alpha=0.5,color='#3B7A57',label='post_score rural')
plt.hist(data2post, bins=nbins,alpha=0.5,color=
'#00FFFF',label='post_score urban')
plt.ylim([0, 4]) # pour ajuster la hauteur aux valeurs des féquences (
à modifier au besoin)
plt.title('Histogramme groupé rural/Urabain')
plt.xlabel('Scores')
plt.ylabel('Fréquences')
plt.legend(loc='upper right')
plt.show()
plt.cla()
plt.clf()
plt.close()
```



```
###### histogramme dans deux cadrans séparés ( un cadre pour rural et un cadre pour urbain)
fig  =plt.figure('myfigure',figsize=(10,6))  #  Nom  et  taille  de  la figure largeur/hauteur
# Cadran rural
ax = fig.add_subplot(121)
nbins =5
plt.hist(data1pre,                                          bins=nbins, alpha=0.5,color='#EDD834',label='pre_score rural')
plt.hist(data1mid,                                          bins=nbins, alpha=0.5,color='#AF002A',label='mid_score rural')
plt.hist(data1post,                                         bins=nbins, alpha=0.5,color='#3B7A57',label='post_score rural')
plt.ylim([0, 5] )
ax.set_ylabel('Fréquences') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Histogram rural') #Titre du cadran
plt.legend(loc='upper right') # à enlever si pas nécessaire
# Cadran urbain
ax = fig.add_subplot(122)
nbins =5
plt.hist(data2pre,                          bins=nbins,alpha=0.5,color= '#887E43',label='pre_score urban')
plt.hist(data2mid,                          bins=nbins,alpha=0.5,color= '#F0F8FF',label='mid_score urban')
plt.hist(data2post,                         bins=nbins,alpha=0.5,color= '#00FFFF',label='post_score urban')
plt.ylim([0, 5] )
ax.set_ylabel('Fréquences') # Titre de l'axe y
ax.set_xlabel('Values')
ax.set_title('Histogram urbain') #Titre du cadran
plt.legend(loc='upper right') # à enlever si pas nécessaire
plt.show()
plt.cla()
plt.clf()
plt.close()
```



Quatrième cas : histogramme de chaque variable sur le milieu rural et urbain dans 6 cadrans différents

Pour cela, il faut utiliser les 6 listes prédémment créees et construire 6 subplots avec chacun des listes et greffer à un cadre nommée comme myfigue comme prédemment.

6.3. Les diagrammes circulaires

6.3.1. Diagramme circulaire simple

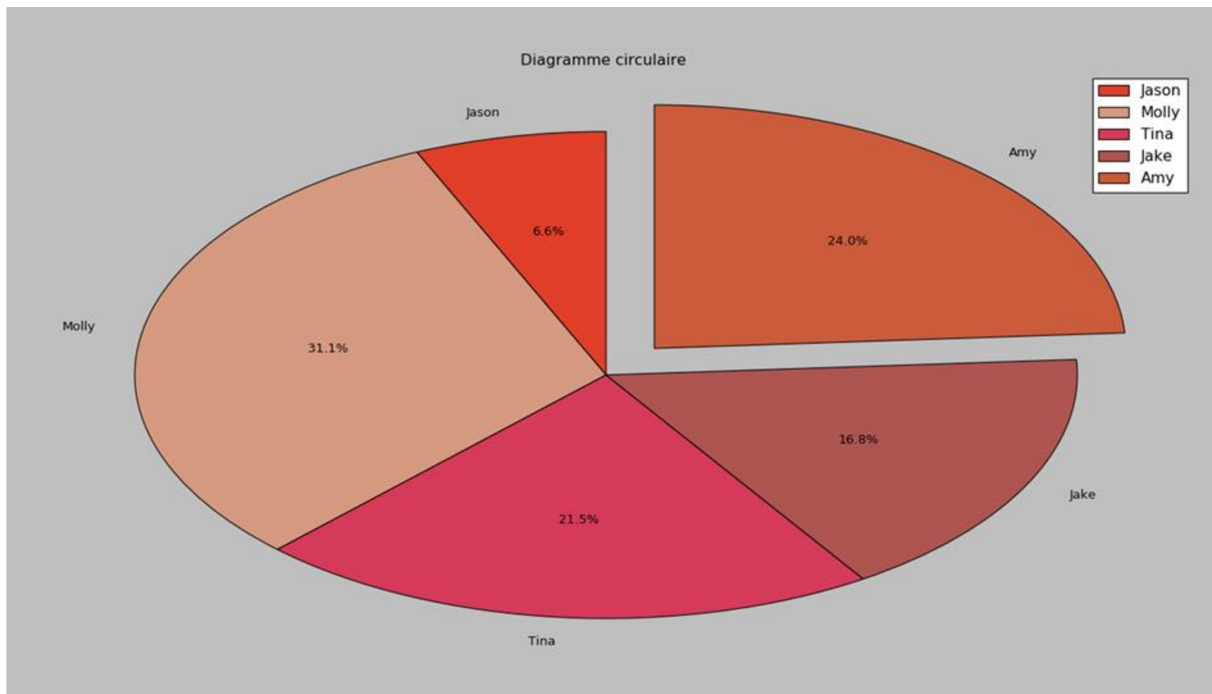
Soit la table de donnée suivante:

```
raw_data = {'officer_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
        'jan_arrests': [4, 24, 31, 2, 3],
        'feb_arrests': [25, 94, 57, 62, 70],
        'march_arrests': [5, 43, 23, 23, 51]}
import pandas as pd
df = pd.DataFrame(raw_data, columns = ['officer_name', 'jan_arrests', 'feb_arrests', 'march_arrests'])
print(df)
```

On souhaite faire un diagramme circulaire pour le nombre total d'arrestations effectuées par les officiers.

```
# D'abord, on calcule le nombre total total d'arrestations:
```

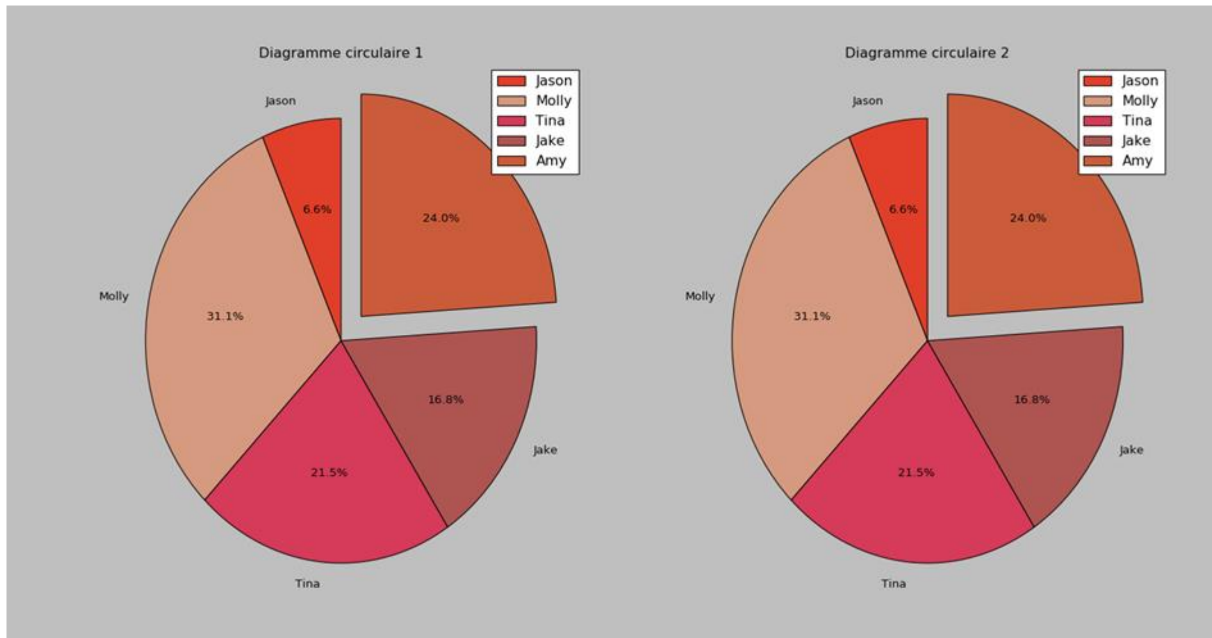
```
df['total_arrests'] = df['jan_arrests'] + df['feb_arrests'] +
df['march_arrests']
## Préparation du graphique
import matplotlib.pyplot as plt
colors = ["#E13F29", "#D69A80", "#D63B59", "#AE5552", "#CB5C3B",
"#EB8076", "#96624E"]
# Crétaion du graphique
fig, ax = plt.subplots(figsize=(20,10)) # taille du graphique
largeur/hauteur
plt.pie(   df['total_arrests'],
           labels=df['officer_name'],# il faut désactiver label si on
ne veut pas faire apparaitre les labels)
           shadow=False,
           colors=colors,# ajout des couleurs
           explode=(0, 0, 0, 0, 0.15), # permet le decoupage du
camembert.
           startangle=90,  # l'angle commence à 90%
           autopct='%1.1f%%',# affichage des pourcentages
      )
ax.set_title('Diagramme circulaire ') #Titre du cadran
plt.legend(df['officer_name'],loc='upper right')
# plt.legend(loc='upper right') # Le label de l'hist tient lieu du
légende si aucune valeur n'est indiquée dans legend
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.3.2. Regrouper plusieurs diagrammes circulaires dans un seul graphique

NB: Tout comme pour les histogrammes et les barres, on peut effectuer les graphiques en cadrans séparés pour les pie. Il suffit de suivre la même démarche en ajoutant les subplots. Voir exemple ci-dessous:

```
### Définition du cadre général:
colors = ["#E13F29", "#D69A80", "#D63B59", "#AE5552", "#CB5C3B", "#EB8076", "#96624E"]
fig =plt.figure('myfigure',figsize=(20,10)) # Nom et taille de la figure largeur/hauteur
############## Premier cadran ( premier graph pie)
ax = fig.add_subplot(121)
plt.pie(   df['total_arrests'],
           labels=df['officer_name'], # il faut désactiver label si on ne veut pas faire apparaitre les labels
           shadow=False,
           colors=colors,
           explode=(0, 0, 0, 0, 0.15),
           startangle=90,
           autopct='%1.1f%%',
      )

ax.set_title('Diagramme circulaire 1') #Titre du cadran
plt.legend(df['officer_name'],loc='upper right')
# plt.legend(loc='upper right')
################### deuxième cadran  (deuxième graph pie)
ax = fig.add_subplot(122)
plt.pie(   df['total_arrests'], # on a pris la même variable ( à titre d'exemple)
           labels=df['officer_name'], # il faut désactiver label si on ne veut pas faire apparaitre les labels
           shadow=False,
           colors=colors,
           explode=(0, 0, 0, 0, 0.15),
           startangle=90,
           autopct='%1.1f%%',
      )
ax.set_title('Diagramme circulaire 2') #Titre du cadran
plt.legend(df['officer_name'],loc='upper right')
# plt.legend(loc='upper right')
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.4. Les diagrammes de fréquences

En pratique, le diagramme des fréquences est un graphique en barres dont les valeurs sont les fréquences (absolues ou relatives) de chaque modalité.

Pour construire ce type de graphique, il faut au préalable construire une liste contenant les fréquences de chaque modalités. Pour construire cette liste, on peut utiliser la fonction value_counts de panda pour generer les modalités et leurs fréquence. Ensuite, on utilise la fonction dist() pour convertir cet objet en liste. Une fois que cette liste des valeurs est obtenue on suit les démarches présenté dans le cas d'un graphique en barres simple où l'axe des abscicess est labelisé par les modalités.

Exemple :

Soit la table de données suivante (qui fournit les réponses à une enquête de satisfaction dans un hotel, voir le fichier word pour les questions et modalités).

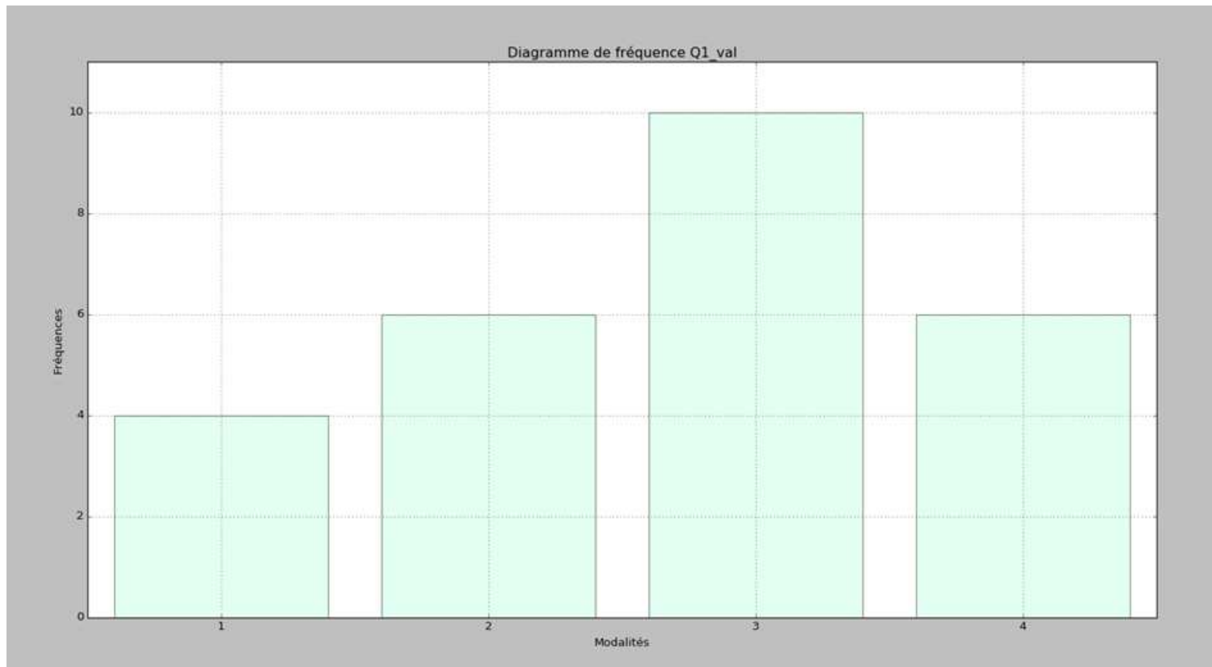
```
df = pandas.read_csv('QUESTIONNAIRE_ACCEUIL2.csv',sep=';', header=0,
encoding ='Latin-1')
print(df.head())
```

Nous allons utiliser la question Q1 pour faire les graphiques de fréquences. On dispose de deux colonnes pour cette question: La colonne Q1_val qui donne les codage numériques et la colonne Q1_mod qui donne les codes en caractères. On va faire l'expérience sur les deux types de variables.

6.4.1. Diagramme de fréquence sur une variable qualitative à codage numérique (ou sur une variable quantitative)

On va considérer la variable Q1_val

```
x=pandas.value_counts(df['Q1_val']) # comptage de la fréquence de chaque modalité
dico=dict(x) # Transformer x en un dictionnaire
print(dico)
# Transformation du dictionnaire en dataframe
col=dico.keys()   # récupérer les clés du dictionnaires. Ces clés représentent les modalités analysées. Elles seront les variables dans le dataframe à créer.
print(col)
df = pandas.DataFrame(dico,columns=col, index=[0]) # index=[0] doit être spécifié lorsque les valeurs du dictionnaire ne sont pas encadrés par [].
freq_values = [df[1].max(), df[2].max(), df[3].max(),df[4].max()] # les 1 , 2, 3 correspondent aux valeurs (il faut connaitre le nombre total de modalités)
print(freq_values)
#Représentation graphique
# cadre du graphique
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(10,5))
x_pos = list(range(len(col))) # Création de l'axe des barres
plt.bar(x_pos,freq_values,  align='center',  color='#C9FFE5',alpha=0.5)
# Création du graphique
plt.grid() # ajouter des grid (grilles)
max_y = max(freq_values) #Graduation de l'axe y
plt.ylim([0, max_y* 1.1])
plt.ylabel('Fréquences') # labeliser les axes
plt.xlabel('Modalités')
plt.xticks(x_pos, col)
plt.title('Diagramme de fréquence Q1_val')
plt.show()
plt.cla()
plt.clf()
plt.close()
```



Pour afficher les valeurs des fréquences sur le graphique, il faut suivre la méthode discutée avec les graphique en barres avec la fonction autolabel()

6.4.2. Diagramme de fréquence sur une variable qualitative à codage en caractères

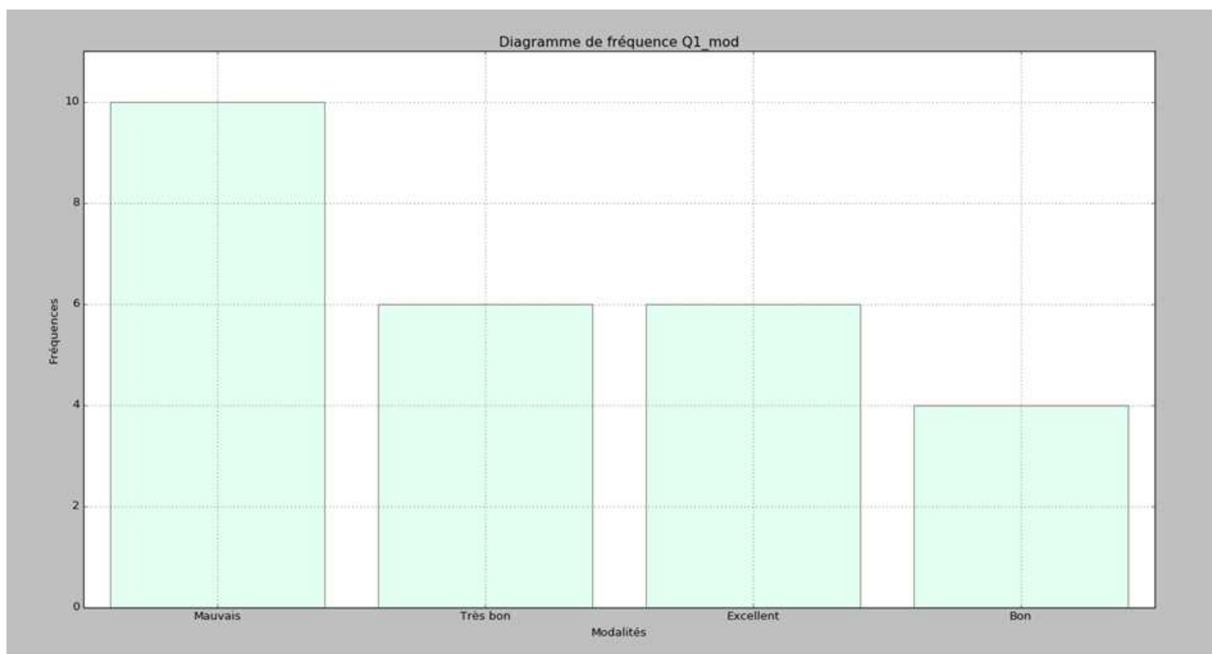
Soit la table :

```
df = pandas.read_csv('QUESTIONNAIRE_ACCEUIL2.csv',sep=';', header=0, encoding ='Latin-1')
print(df.head())
```

On va considérer la variable Q1_mod

```
x=pandas.value_counts(df['Q1_mod']) # comptage de la fréquence de chaque modalité
dico=dict(x) # Transformer x en un dictionnaire
print(dico)
# Transformation du dictionnaire en dataframe
col=dico.keys()  # récupérer les clés du dictionnaires. Ces clés représentent les modalités analysées. Elles seront les variables dans le dataframe à créer.
print(col)
df = pandas.DataFrame(dico,columns=col, index=[0]) # index=[0] doit être spécifié lorsque les valeurs du dictionnaire ne sont pas encadré par [].
```

```
freq_values   =   [df['Bon'].max(),   df['Mauvais'].max(),   df['Très
bon'].max(),df['Excellent'].max()]
print(freq_values)
#Représentation graphique
# cadre du graphique
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(10,5))
x_pos = list(range(len(col))) # Création de l'axe des barres
plt.bar(x_pos,freq_values,  align='center',  color='#C9FFE5',alpha=0.5)
# Création du graphique
plt.grid() # ajouter des grid (grilles)
max_y = max(freq_values) #Graduation de l'axe y
plt.ylim([0, max_y* 1.1])
plt.ylabel('Fréquences') # labeliser les axes
plt.xlabel('Modalités')
plt.xticks(x_pos, col)
plt.title('Diagramme de fréquence Q1_mod')
plt.show()
plt.cla()
plt.clf()
plt.close()
```



Là aussi, pour afficher les valeurs des fréquences sur le graphique, il faut suivre la méthode discutée avec les graphiques en barres avec la fonction autolabel()

6.5. Les graphiques en nuages de points

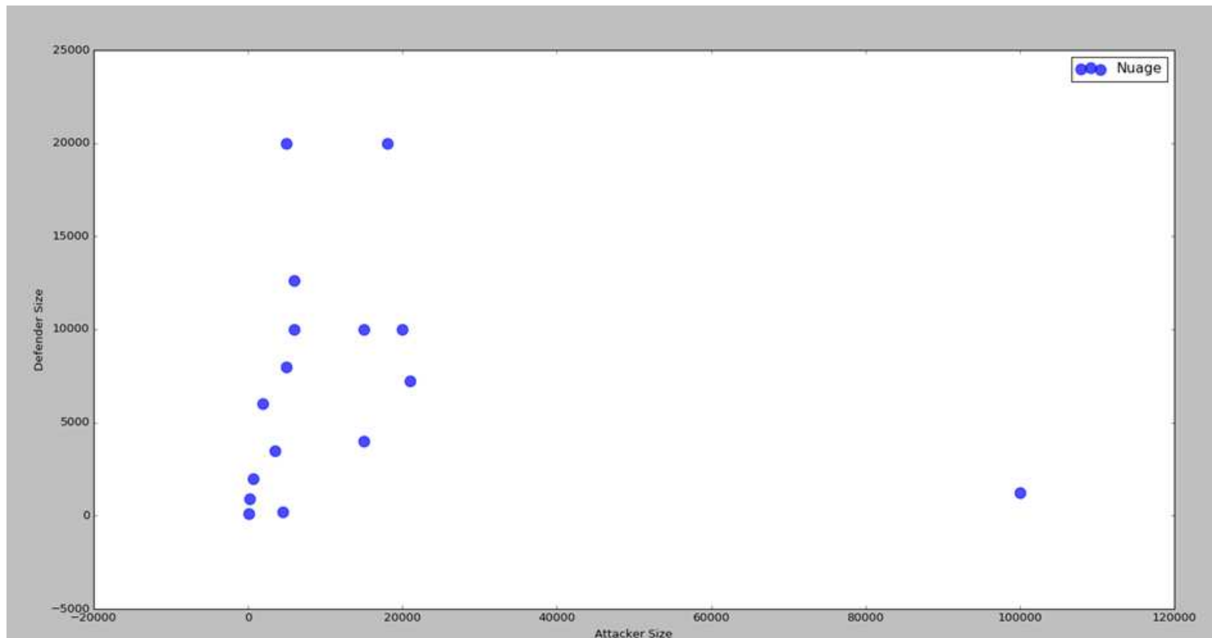
6.5.1. Graphique simple de nuage de points

Soit la table de données suivante:

```
df = pandas.read_csv('5kings_battles_v1.csv')
print(df.head())
```

Nous allons réaliser un nuage de points entre les variables attacker_size et defender_siz

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots(figsize=(20,10)) # taille du graphique largeur/hauteur
plt.scatter(  df['attacker_size'],
              df['defender_size'], # attacker_size l'axe des y
              marker='o', # le marqueur
              color='b',# la couleur bleur
              alpha=0.7,# valeur de alpha
              s = 124,  # taille # On peut aussi indique le nom d'une variable pour que les tailles des point soient proportionlles. Ainsi on fait s=df.age
              label='Nuage' # label du graphique
            )
##plt.xlim([min(df['attacker_size'])-1000, max(df['attacker_size'])+1000])
##plt.ylim([min(df['defender_size'])-1000, max(df['defender_size'])+1000])
plt.ylabel('Defender Size') # y label
plt.xlabel('Attacker Size') # x label
plt.legend(loc='upper right') # légende. Elle est égale au label si aucune valeur n'est indiquée
# On pouvait aussi utiliser les option ax.
#ax.set_ylabel('Fréquences') # Titre de l'axe y
#ax.set_xlabel('Values of x')
#ax.set_title('Histogram urbain') #Titre du cadran
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.5.2. Labéliser les valeurs dans un nuage de points

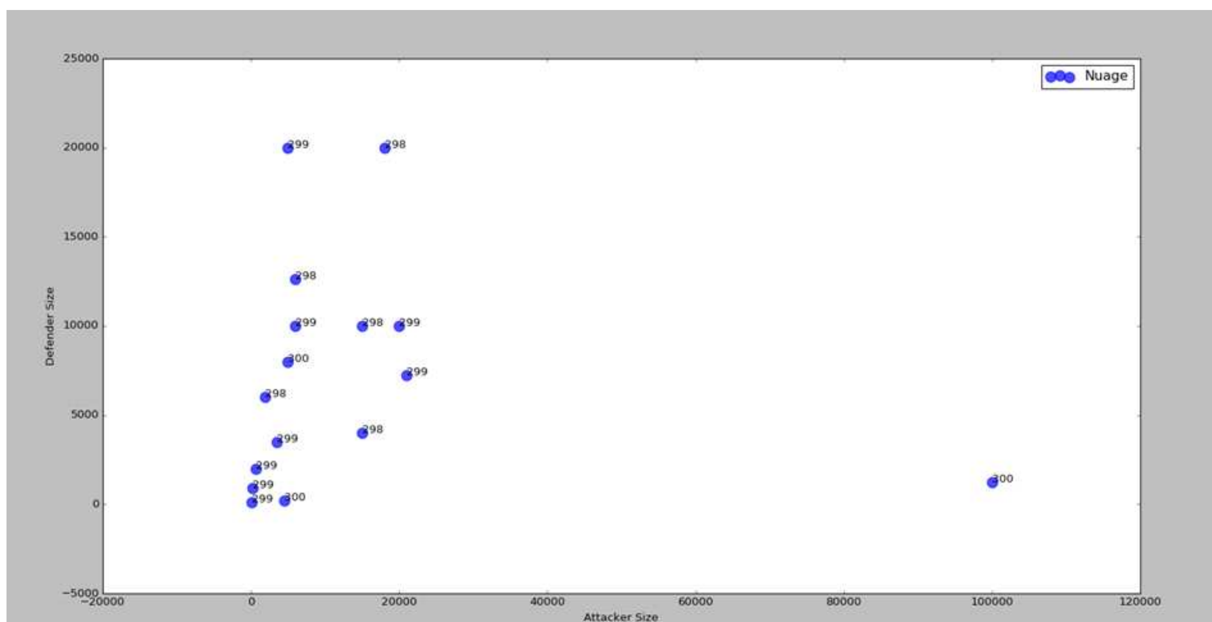
Soit le graphique suivant construit à partir de la table df comme précédemment:

```
fig, ax = plt.subplots(figsize=(20,10))
plt.scatter(  df['attacker_size'],
              df['defender_size'],
              marker='o',
              color='b',
              alpha=0.7,
              s = 124,
              label='Nuage'
            )
plt.ylabel('Defender Size')
plt.xlabel('Attacker Size')
plt.legend(loc='upper right')
#plt.show()
plt.cla()
plt.clf()
plt.close()
```

On va labeliser les valeurs du nuage de point par la valeur de la variable year. Alors, on ajoute la commande annotate comme suit:

```
list_year=df['year'] # on récupère les valeur de year dans un liste
nommée list_year
print(list_year)
fig, ax = plt.subplots(figsize=(20,10))
ax.scatter(  df['attacker_size'], # avec ax
```

```
            df['defender_size'],
            marker='o',
            color='b',
            alpha=0.7,
            s = 124,
            label='Nuage'
         )
# Ajout des labels au points avec les valeurs de list_year
for  i,  mytxt  in  enumerate(list_year):  #  On  fait  annotate  pour
l'ensemble des valeurs de list_year
    ax.annotate(mytxt,
(df['attacker_size'][i],df['defender_size'][i]))

plt.ylabel('Defender Size')
plt.xlabel('Attacker Size')
plt.legend(loc='upper right')
plt.show()
plt.cla()
plt.clf()
plt.close()
```



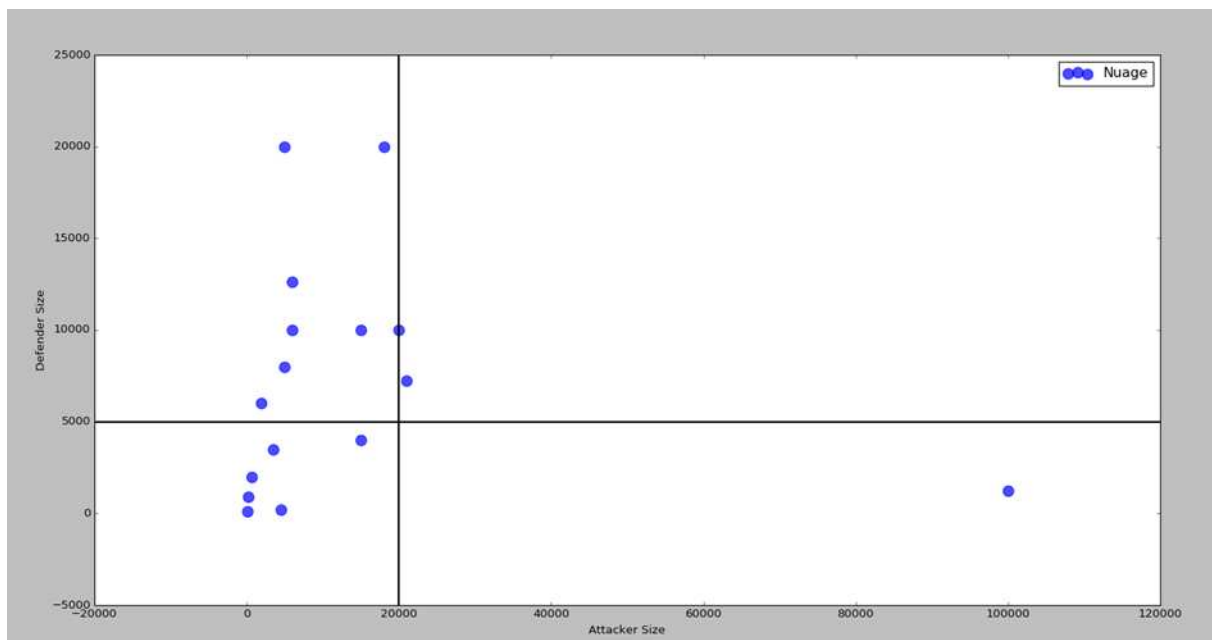
6.5.3. Ajouter une ligne verticale ou horizontale à un nuage de point

Pour ajouter des lignes verticales (ou horizontales) à un graph, on utilise les options axhline et axvline.

Exemple:

```
fig, ax = plt.subplots(figsize=(20,10))
ax.scatter(  df['attacker_size'], # avec ax
```

```
                df['defender_size'],
                marker='o',
                color='b',
                alpha=0.7,
                s = 124,
                label='Nuage'
            )
plt.axhline(y=5000, xmin=-2000, xmax=12000, linewidth=2, color = 'k')
# HORIZONTAL
plt.axvline(x=20000, ymin=-5000, ymax = 25000, linewidth=2, color='k')
# VERTICAL
plt.ylabel('Defender Size')
plt.xlabel('Attacker Size')
plt.legend(loc='upper right')
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.5.4. Nuage de points selon des catégories (représentation dans le même cadran)

Soit la table de données suivante:

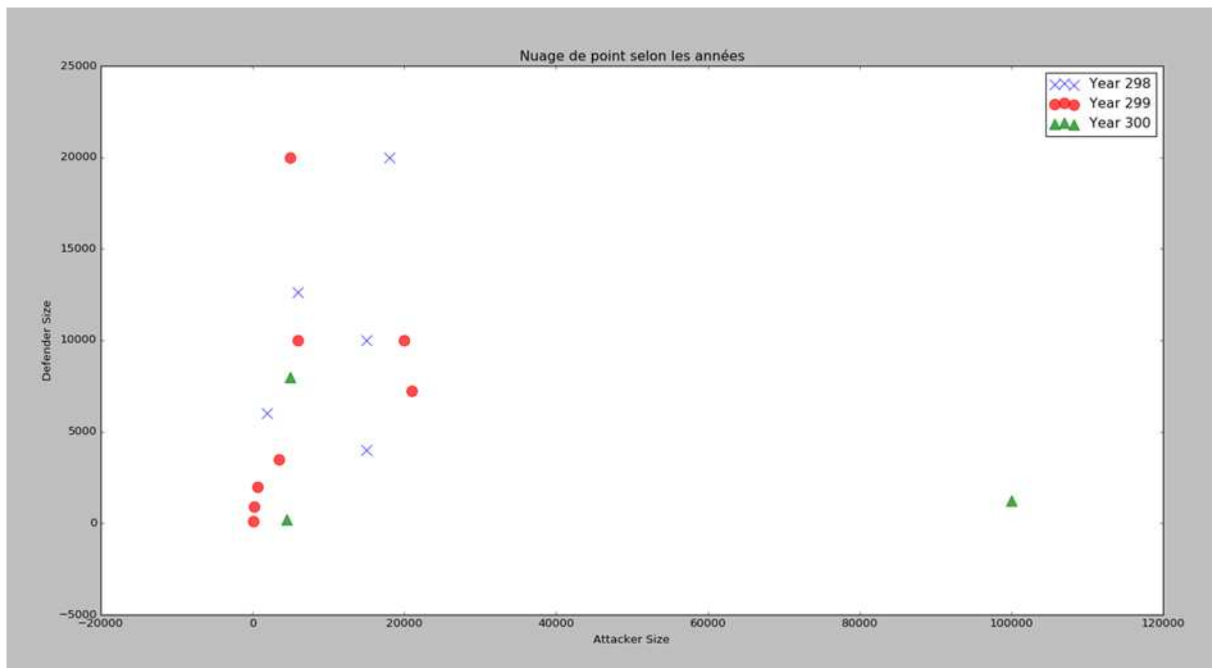
```
df = pandas.read_csv('5kings_battles_v1.csv')
print(df.head())
```

Nous allons réaliser un nuage de points entre les variables attacker_size et defender_siz

Effectuons un nuage de point pour chaque valeur de la valeur year et représentons ces trois nuages dans le même cadran On a:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
#plt.figure(figsize=(10,8))
fig, ax = plt.subplots(figsize=(20,10)) # taille du graphique largeur/hauteur
# Graphique pour 298
plt.scatter(df['attacker_size'][df['year'] == 298], # attacker size in year 298 as the x axis
            df['defender_size'][df['year'] == 298], # attacker size in year 298 as the y axis
            marker='x',
            color='b',
            alpha=0.7,
            s = 124,
            label='Year 298'
            )
 # Graphique pour 299
plt.scatter(df['attacker_size'][df['year'] == 299],# attacker size in year 299 as the x axis
            df['defender_size'][df['year'] == 299],# defender size in year 299 as the y axis
            marker='o',
            color='r',
            alpha=0.7,
            s = 124,
            label='Year 299'
            )
# Graphique pour 300
plt.scatter(df['attacker_size'][df['year'] == 300],# attacker size in year 300 as the x axis
            df['defender_size'][df['year'] == 300],# defender size in year 300 as the y axis
            marker='^',
            color='g',
            alpha=0.7,
            s = 124,
            label='Year 300'
            )
# Mise en forme
# Limite de la zone de graphique
##plt.xlim([min(df['attacker_size'])-1000,
max(df['attacker_size'])+1000])
##plt.ylim([min(df['defender_size'])-1000,
max(df['defender_size'])+1000])
plt.title('Nuage de point selon les années') # Chart title
```

```
plt.ylabel('Defender Size') # y label
plt.xlabel('Attacker Size') # x label
plt.legend(loc='upper right') # legend.
# On pouvait aussi utiliser les option ax.
#ax.set_ylabel('Fréquences') # Titre de l'axe y
#ax.set_xlabel('Values of x')
#ax.set_title('Histogram urbain') #Titre du cadran
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.5.5. Nuage de points selon des catégories (représentation dans des cadrans différents)

Soit la table de données suivante:

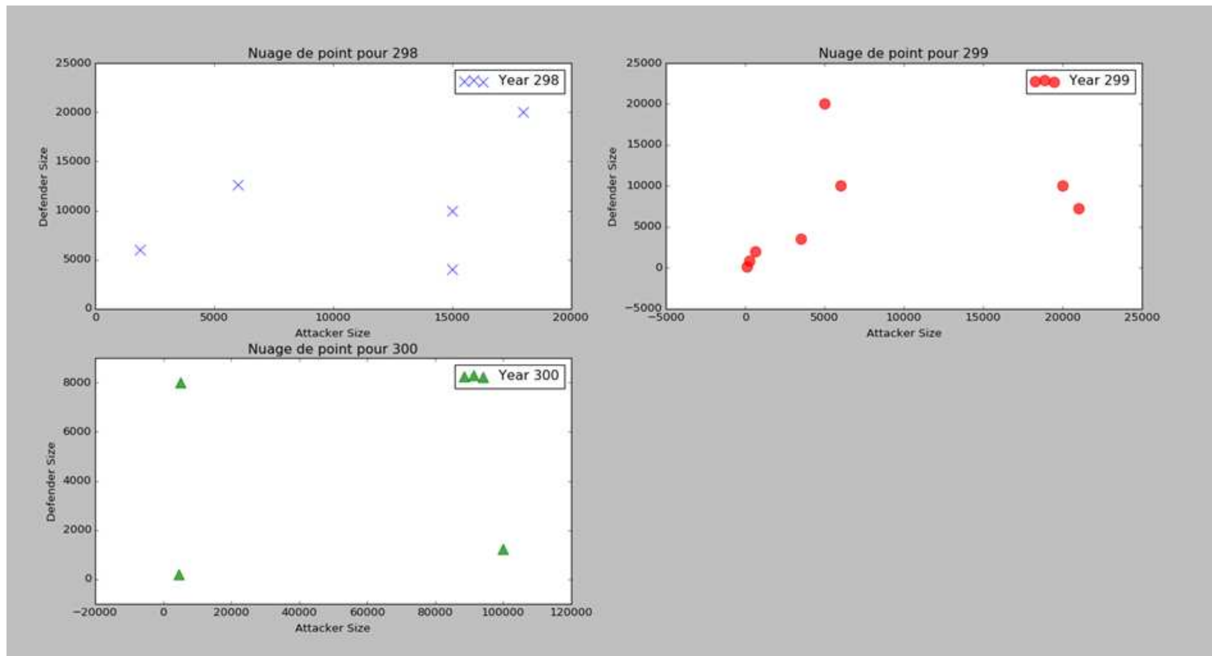
```
df = pandas.read_csv('5kings_battles_v1.csv')
print(df.head())
```

Nous allons réaliser un nuage de points entre les variables attacker_size et defender_siz

Effectuon un nuage de point pour chaque valeur de la valeur year et représentons ces trois nuage dans trois cadran différents. On a:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
fig  =plt.figure('myfigure',figsize=(20,10))  #  Nom  et  taille  de  la
figure largeur/hauteur

###### Premier cadran ( premier scatter pour year= 298)
ax = fig.add_subplot(221)
plt.scatter(df['attacker_size'][df['year'] == 298], # attacker size in
year 298 as the x axis
            df['defender_size'][df['year'] == 298], # attacker size in
year 298 as the y axis
            marker='x',
            color='b',
            alpha=0.7,
            s = 124,
            label='Year 298'
            )
#plt.xlim([min(df['attacker_size'][df['year']         ==         298]),
max(df['attacker_size'][df['year'] == 298])])
#plt.ylim([min(df['defender_size'][df['year']         ==         298]),
max(df['defender_size'][df['year'] == 298])])
plt.title('Nuage de point pour 298') # Chart title
plt.ylabel('Defender Size') # y label
plt.xlabel('Attacker Size') # x label
plt.legend(loc='upper  right')  #  legend.  Elle  est  égale  au  label  si
aucune valeur n'est indiquée
# On pouvait aussi utiliser les option ax.
#ax.set_ylabel('Fréquences') # Titre de l'axe y
#ax.set_xlabel('Values of x')
#ax.set_title('Histogram urbain') #Titre du cadran

 ###### Deuxième cadran (pour year= 299)
ax = fig.add_subplot(222)
plt.scatter(df['attacker_size'][df['year'] == 299],# attacker size in
year 299 as the x axis
            df['defender_size'][df['year'] == 299],# defender size in
year 299 as the y axis
            marker='o',
            color='r',
            alpha=0.7,
            s = 124,
            label='Year 299'
            )
#plt.xlim([min(df['attacker_size'][df['year']         ==         299]),
max(df['attacker_size'][df['year'] == 299])])
#plt.ylim([min(df['defender_size'][df['year']         ==         299]),
max(df['defender_size'][df['year'] == 299])])
plt.title('Nuage de point pour 299') # Chart title
```

```
plt.ylabel('Defender Size') # y label
plt.xlabel('Attacker Size') # x label
plt.legend(loc='upper right') # legend. Elle est égale au label si aucune valeur n'est indiquée
# On pouvait aussi utiliser les option ax.
#ax.set_ylabel('Fréquences') # Titre de l'axe y
#ax.set_xlabel('Values of x')
#ax.set_title('Histogram urbain') #Titre du cadran

###### troisième cadran (  pour year= 300)
ax = fig.add_subplot(223)
plt.scatter(df['attacker_size'][df['year'] == 300],# attacker size in year 300 as the x axis
            df['defender_size'][df['year'] == 300],# defender size in year 300 as the y axis
            marker='^',
            color='g',
            alpha=0.7,
            s = 124,
            label='Year 300'
            )
# Mise en forme
#plt.xlim([min(df['attacker_size'][df['year'] == 300]), max(df['attacker_size'][df['year'] == 300])])
#plt.ylim([min(df['defender_size'][df['year'] == 300]), max(df['defender_size'][df['year'] == 300])])
plt.title('Nuage de point pour 300') # Chart title
plt.ylabel('Defender Size') # y label
plt.xlabel('Attacker Size') # x label
plt.legend(loc='upper right') # legend. Elle est égale au label si aucune valeur n'est indiquée
# On pouvait aussi utiliser les option ax.
#ax.set_ylabel('Fréquences') # Titre de l'axe y
#ax.set_xlabel('Values of x')
#ax.set_title('Histogram urbain') #Titre du cadran
plt.show()
plt.cla()
plt.clf()
plt.close()
```

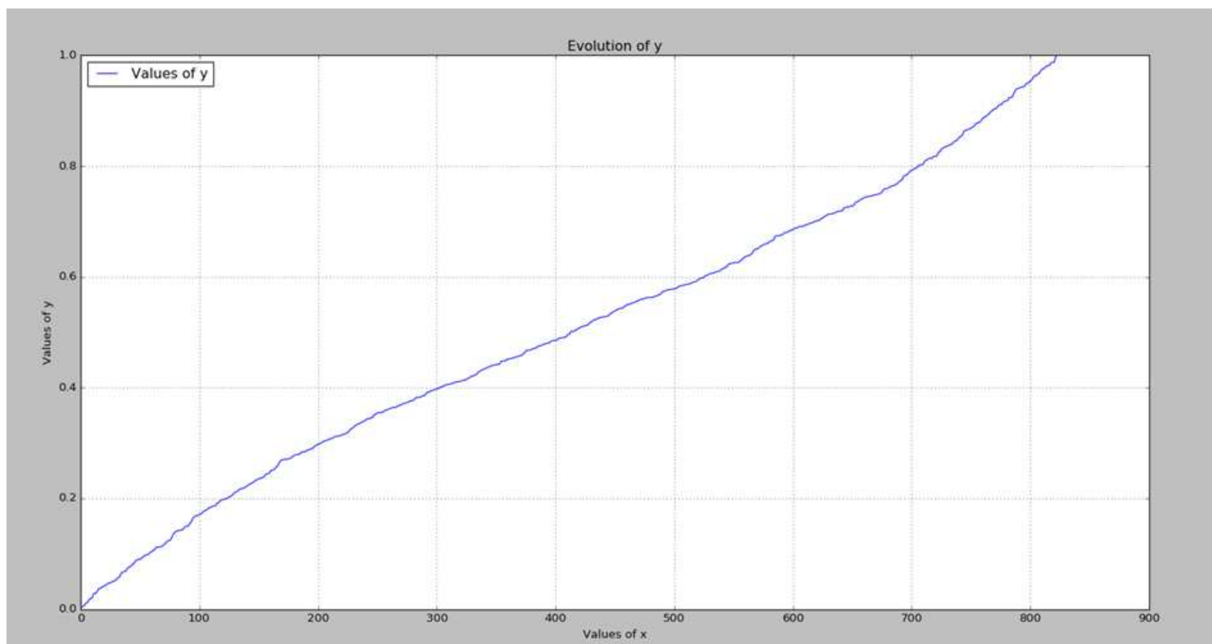


6.6. Graphique en courbe d'évolution (ligne)

6.6.1. Courbe d'évolution d'une seule variable

```
#Soit y une variable définie par la liste suivante :
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(10) # fixation du seed
y = np.random.normal(loc=0.5, scale=0.4, size=1000) # générer y entre
0 et 1
y = y[(y > 0) & (y < 1)] # Garder les valeur supérieur à 0 et
inférieures à 1
y.sort() # Trier les valeurs de la liste du plus petit au plus grand
# Soit x l'axe des abcisse définie par la liste suivante
x = np.arange(len(y))
#  On peut représenter la courbe de y en fonction de x comme suit:
# cadre de base du graphique
fig, ax = plt.subplots(figsize=(10,6)) # taille du graphique
largeur/hauteur
# Tracer
plt.plot(x, y)
ax.set_ylabel('Values of y')
ax.set_xlabel('Values of x')
ax.set_title('Evolution of y') #Titre du graphique
plt.legend(['Values of y'], loc='upper left') # Définition des
légendes
plt.grid()
plt.show()
```

```
plt.cla()
plt.clf()
plt.close()
```



6.6.2. Représenter plusieurs courbes dans un même cadran

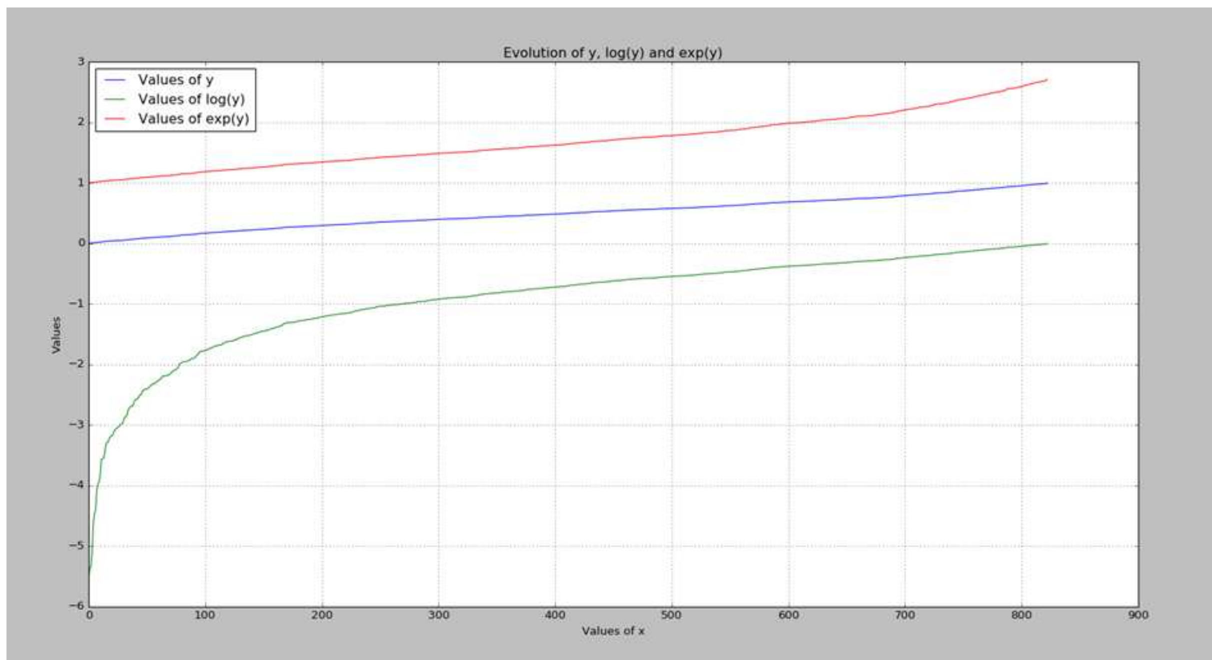
Soit les variables x et y telle que définies pour le graphique précédent. Calculons le log et l'exponentielle de y

```
logy=np.log(y) #  le log
expy=np.exp(y) # exponentielle
```

Maintenant, représention les courbes des trois variables y, logy et expy en fonction de x dans le même cadran. On a :

```
fig, ax = plt.subplots(figsize=(10,6)) # taille du graphique
largeur/hauteur
plt.plot(x, y) # représentation linéaire
plt.plot(x, logy) # représentation en log
plt.plot(x, expy) # représentation en exp
ax.set_ylabel('Values')
ax.set_xlabel('Values of x')
ax.set_title('Evolution of y, log(y) and exp(y)') #Titre du graphique
plt.legend(['Values of y','Values of log(y)','Values of exp(y)'],
loc='upper left')
plt.grid()
plt.show()
plt.cla()
plt.clf()
```

```
plt.close()
```

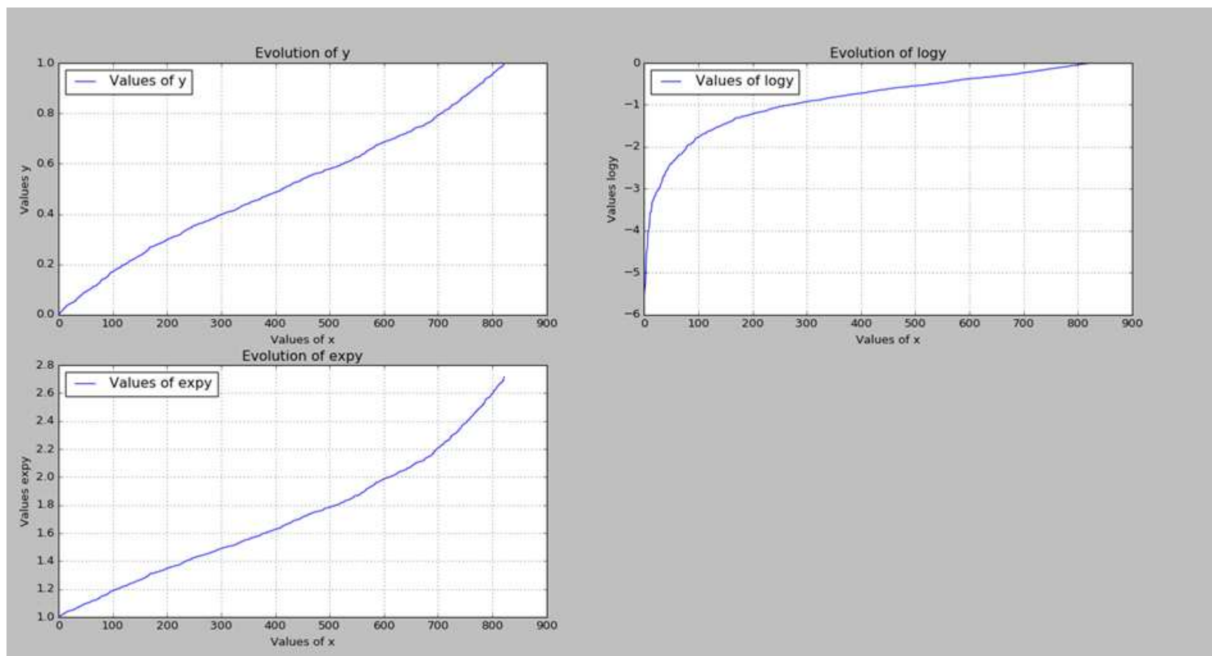


6.6.3. Représenter plusieurs courbes (dans des cadrans différents)

Soit les variables x, y, logy et expy telle que précédement définie. Maintenant, représentons les courbe des trois variables y, logy et expy en fonction de x dans des cadrans différents. On a :

```
fig  =plt.figure('myfigure',figsize=(20,10))  #  Nom  et  taille  de  la
figure largeur/hauteur
###### Premier cadran ( pour y)
ax = fig.add_subplot(221)
plt.plot(x, y) # représentation linéaire
ax.set_ylabel('Values y')
ax.set_xlabel('Values of x')
ax.set_title('Evolution of y') #Titre du graphique
plt.legend(['Values of y'], loc='upper left')
plt.grid()
###### Deuxième cadran ( pour logy)
ax = fig.add_subplot(222)
plt.plot(x, logy) # représentation linéaire
ax.set_ylabel('Values logy')
ax.set_xlabel('Values of x')
ax.set_title('Evolution of logy') #Titre du graphique
plt.legend(['Values of logy'], loc='upper left')
plt.grid()
###### Troisième cadran ( pour expy)
ax = fig.add_subplot(223)
plt.plot(x, expy) # représentation linéaire
ax.set_ylabel('Values expy')
```

```
ax.set_xlabel('Values of x')
ax.set_title('Evolution of expy') #Titre du graphique
plt.legend(['Values of expy'], loc='upper left')
plt.grid()
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.7. Graphiques en box-plot

6.7.1. Box plot simple sur une seule variable

Soit la table de données suivante

```
raw_data = {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy',
'Kader', 'Jean' , 'Philipe', 'Paul', 'Pierre', 'Jacques'],
        'pre_score': [4, 24, 31, 2, 3, 4 , 10 , 25 , 40, 30, 18],
        'mid_score': [25, 94, 57, 62, 70, 12 , 24 , 25 ,64, 28, 20],
        'post_score': [5, 43, 23, 23, 51, 14 , 27 , 26, 70, 80 ,30],
        'Area': ['Rural', 'Urban', 'Rural', 'Urban', 'Urban', 'Rural',
'Urban' , 'Rural', 'Urban', 'Urban', 'Rural']}
df = pandas.DataFrame(raw_data, columns = ['first_name', 'pre_score',
'mid_score', 'post_score', 'Area'])
print(df)
```

Le but est de faire des box-plots sur les variables pre_score, mid_score et post_score

```
# Créons d'abord les liste de ces trois variables
import numpy as np
import matplotlib as mpl
```

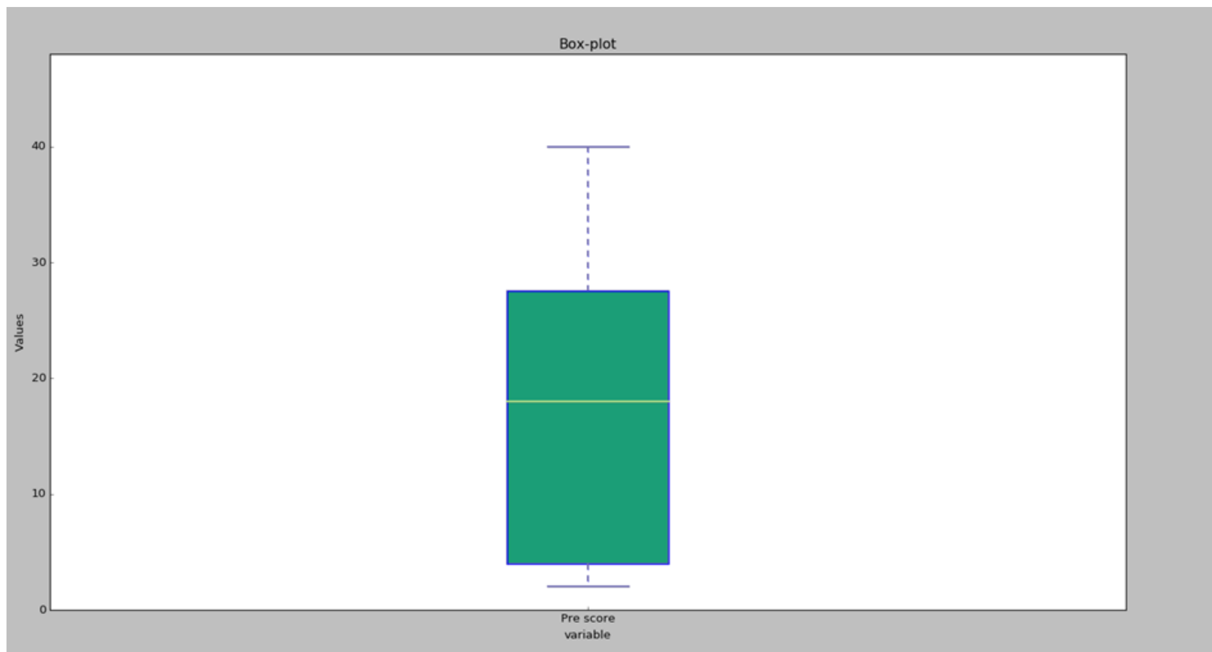
```
import matplotlib.pyplot as plt
data1 = df['pre_score']
data2 = df['mid_score']
data3 = df['post_score']
```

Box-plot simple sur une variable:

Faisons un box-plot sur la variable pre_score avec la liste data1. On a :

```
fig, ax = plt.subplots(figsize=(20, 10))
# Représentation graphique
plt.boxplot(data1)
bp = ax.boxplot(data1) # On crée un objet bp qui récupère tous les attribus des axes afin de pouvoir les mettre en forme après
print(bp) # l'objet dictionnaire bp contient plusieurs attributs (boxes,whiskers,means,medians,caps et fliers). On a utiliser ces attributs pour mettre en forme le graphique
#Prépartion de l'ajout des couleurs
bp = ax.boxplot(data1, patch_artist=True) ## add patch_artist=True option to ax.boxplot() to get fill color
#Ajout des couleurs aux boxs
for box in bp['boxes']: ## change outline color, fill color and linewidth of the boxes
    box.set( color='#7570b3', linewidth=2) # change outline color
    box.set( facecolor = '#1b9e77' ) # change fill color
## #Ajout des couleurs aux whiskers
for whisker in bp['whiskers']:
    whisker.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux caps
for cap in bp['caps']:
    cap.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux medians
for median in bp['medians']:
    median.set(color='#b2df8a', linewidth=2)
## #Ajout des couleurs aux fliers
for flier in bp['fliers']:
    flier.set(marker='o', color='#e7298a', alpha=0.5)
# Changer les labels de l'axe des abscisses
ax.set_xticklabels(['Pre score'])
# Enlever les graduations à droite et en haut
ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()
# libellé des axes
ax.set_ylabel('Values')
ax.set_xlabel('variable')
ax.set_title('Box-plot') #Titre du graphique
plt.ylim([0, data1.max()*1.20])
#plt.ylim([0, max(data1)*1.20]) # equivalent
plt.show()
plt.cla()
```

```
plt.clf()
plt.close()
```



6.7.2. Box plots par catégorie ou pour plusieurs variables (representés dans le même cadran)

Avant de faire ce type de représentation, il faut d'abord créer des listes pour chaque catégorie (ou pour chaque variable). Ensuite regrouper ces listes dans une nouvelle liste. On forme ainsi une liste des listes. Avec cette liste des listes, on peut directement représenter un graphique sans avoir besoin de faire des subplots liste par liste. Par exemple, considérons les trois liste définies par data1, data2 et data3 (précédemment définies). On va regrouper ces trois listes dans une nouvelle liste nommée data_to_plot

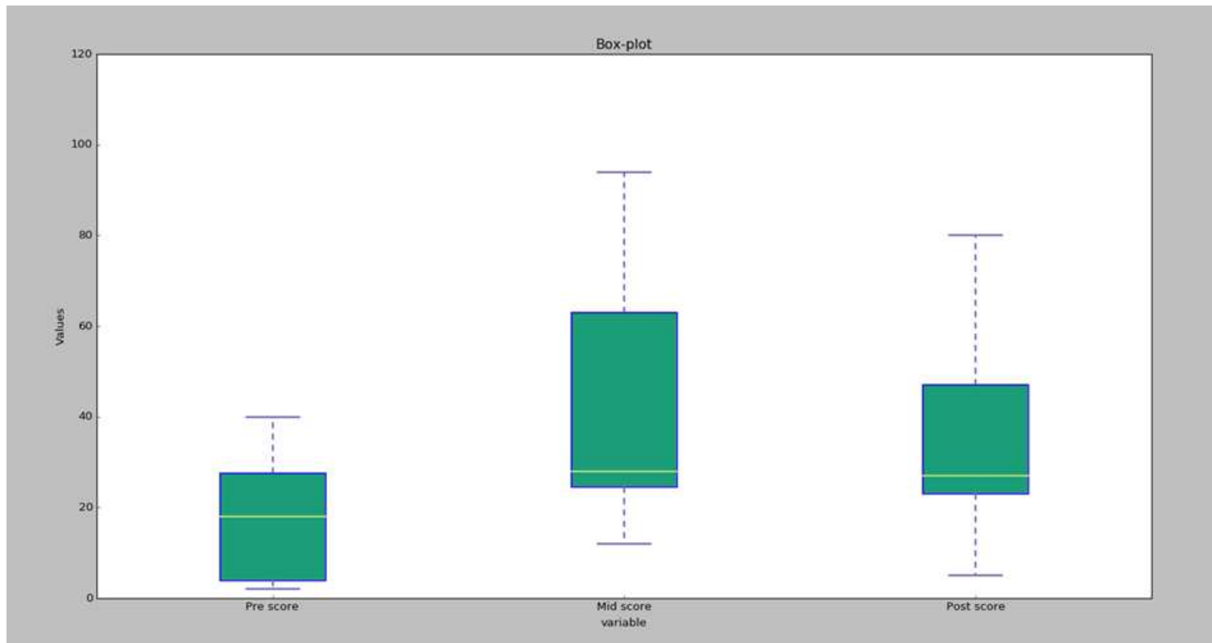
```
data_to_plot = [data1, data2, data3]
```

On utilise ainsi cette liste pour représenter directement les box-plot (voir ci-dessous).

NB: Il faut noter qu'on pouvait représenter liste par liste. Mais cela aurait nécessiter de définir chaque fois la position du box plot en paramétrant les axes à chaque fois. Mais avec une liste des liste, on a pas besoin d'un tel paramétrage.

```
#représentation
fig, ax = plt.subplots(figsize=(20, 10))
# Représentation graphique
plt.boxplot(data_to_plot)
bp = ax.boxplot(data_to_plot) # On crée un objet bp qui récupère tous
les attribus des axes afin de pouvoir les mettre en forme après
```

```
print(bp) # l'objet dictionnaire bp contient plusieurs attributs (boxes,whiskers,means,medians,caps et fliers). On a utiliser ces attributs pour mettre en forme le graphique
#Prépartion de l'ajout des couleurs
bp = ax.boxplot(data_to_plot, patch_artist=True) ## add patch_artist=True option to ax.boxplot() to get fill color
#Ajout des couleurs aux boxs
for box in bp['boxes']: ## change outline color, fill color and linewidth of the boxes
    box.set( color='#7570b3', linewidth=2) # change outline color
    box.set( facecolor = '#1b9e77' ) # change fill color
## #Ajout des couleurs aux whiskers
for whisker in bp['whiskers']:
    whisker.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux caps
for cap in bp['caps']:
    cap.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux medians
for median in bp['medians']:
    median.set(color='#b2df8a', linewidth=2)
## #Ajout des couleurs aux fliers
for flier in bp['fliers']:
    flier.set(marker='o', color='#e7298a', alpha=0.5)
# Changer les labels de l'axe des abscisses
ax.set_xticklabels(['Pre score','Mid score','Post score'])
# Enlever les graduations à droite et en haut
ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()
# libellé des axes
ax.set_ylabel('Values')
ax.set_xlabel('variable')
ax.set_title('Box-plot') #Titre du graphique
plt.ylim([0, 120])
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.7.3. Box plots par catégorie ou pour plusieurs variables (représentés dans des cadrans différents)

Pour cette représentation , on utilise liste par liste comme dans les histogrammes, les barres, les pie, les scatter et les lignes.

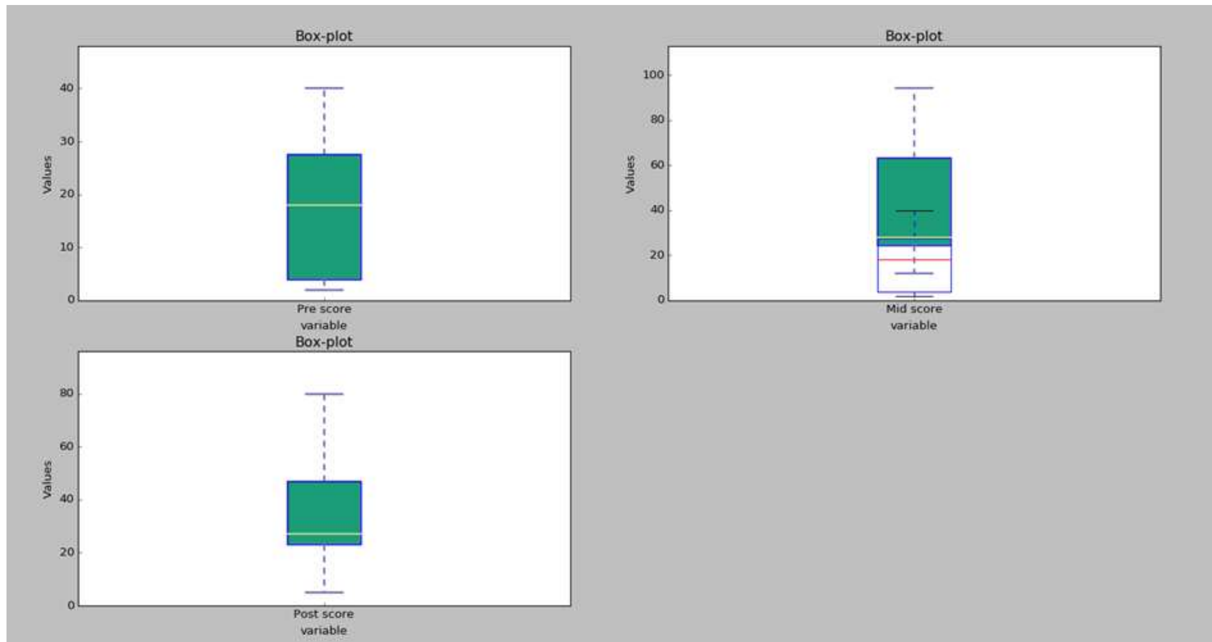
En considérant les trois liste définies pour le graphique précédent, la démarche est la suivante.

```
fig  =plt.figure('myfigure',figsize=(20,10))  #  Nom  et  taille  de  la
figure largeur/hauteur
###### Premier cadran ( pour pre_score avec data1)
ax = fig.add_subplot(221)
# Représentation graphique
plt.boxplot(data1)
bp  =  ax.boxplot(data1)  #  On  crée  un  objet  bp  qui  récupère  tous  les
attribus des axes afin de pouvoir les mettre en forme après
print(bp)  #  l'objet  dictionnaire  bp  contient  plusieurs  attributs
(boxes,whiskers,means,medians,caps  et  fliers).  On  a  utiliser  ces
attributs pour mettre en forme le graphique
#Prépartion de l'ajout des couleurs
bp  =  ax.boxplot(data1,  patch_artist=True)  ##  add  patch_artist=True
option to ax.boxplot() to get fill color
#Ajout des couleurs aux boxs
for  box  in  bp['boxes']:  ##  change  outline  color,  fill  color  and
linewidth of the boxes
    box.set( color='#7570b3', linewidth=2) # change outline color
    box.set( facecolor = '#1b9e77' ) # change fill color
## #Ajout des couleurs aux whiskers
for whisker in bp['whiskers']:
```

```
    whisker.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux caps
for cap in bp['caps']:
    cap.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux medians
for median in bp['medians']:
    median.set(color='#b2df8a', linewidth=2)
## #Ajout des couleurs aux fliers
for flier in bp['fliers']:
    flier.set(marker='o', color='#e7298a', alpha=0.5)
# Changer les labels de l'axe des abscisses
ax.set_xticklabels(['Pre score'])
# Enlever les graduations à droite et en haut
ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()
# libellé des axes
ax.set_ylabel('Values')
ax.set_xlabel('variable')
ax.set_title('Box-plot') #Titre du graphique
plt.ylim([0, max(data1)*1.20]) #
###### Deuxième cadran ( pour mid_score avec data2)
ax = fig.add_subplot(222)
# Représentation graphique
plt.boxplot(data1)
bp = ax.boxplot(data2) # On crée un objet bp qui récupère tous les attribus des axes afin de pouvoir les mettre en forme après
print(bp) # l'objet dictionnaire bp contient plusieurs attributs (boxes,whiskers,means,medians,caps et fliers). On a utiliser ces attributs pour mettre en forme le graphique
#Prépartion de l'ajout des couleurs
bp = ax.boxplot(data2, patch_artist=True) ## add patch_artist=True option to ax.boxplot() to get fill color
#Ajout des couleurs aux boxs
for box in bp['boxes']: ## change outline color, fill color and linewidth of the boxes
    box.set( color='#7570b3', linewidth=2) # change outline color
    box.set( facecolor = '#1b9e77' ) # change fill color
## #Ajout des couleurs aux whiskers
for whisker in bp['whiskers']:
    whisker.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux caps
for cap in bp['caps']:
    cap.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux medians
for median in bp['medians']:
    median.set(color='#b2df8a', linewidth=2)
## #Ajout des couleurs aux fliers
for flier in bp['fliers']:
    flier.set(marker='o', color='#e7298a', alpha=0.5)
```

```
# Changer les labels de l'axe des abscisses
ax.set_xticklabels(['Mid score'])
# Enlever les graduations à droite et en haut
ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()
# libellé des axes
ax.set_ylabel('Values')
ax.set_xlabel('variable')
ax.set_title('Box-plot') #Titre du graphique
plt.ylim([0, max(data2)*1.20]) #
###### troisième cadran ( pour post_score avec data3)
ax = fig.add_subplot(223)
# Représentation graphique
plt.boxplot(data3)
bp = ax.boxplot(data3) # On crée un objet bp qui récupère tous les attribus des axes afin de pouvoir les mettre en forme après
print(bp) # l'objet dictionnaire bp contient plusieurs attributs (boxes,whiskers,means,medians,caps et fliers). On a utiliser ces attributs pour mettre en forme le graphique
#Prépartion de l'ajout des couleurs
bp = ax.boxplot(data3, patch_artist=True) ## add patch_artist=True option to ax.boxplot() to get fill color
#Ajout des couleurs aux boxs
for box in bp['boxes']: ## change outline color, fill color and linewidth of the boxes
    box.set( color='#7570b3', linewidth=2) # change outline color
    box.set( facecolor = '#1b9e77' ) # change fill color
## #Ajout des couleurs aux whiskers
for whisker in bp['whiskers']:
    whisker.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux caps
for cap in bp['caps']:
    cap.set(color='#7570b3', linewidth=2)
## #Ajout des couleurs aux medians
for median in bp['medians']:
    median.set(color='#b2df8a', linewidth=2)
## #Ajout des couleurs aux fliers
for flier in bp['fliers']:
    flier.set(marker='o', color='#e7298a', alpha=0.5)
# Changer les labels de l'axe des abscisses
ax.set_xticklabels(['Post score'])
# Enlever les graduations à droite et en haut
ax.get_xaxis().tick_bottom()
ax.get_yaxis().tick_left()
# libellé des axes
ax.set_ylabel('Values')
ax.set_xlabel('variable')
ax.set_title('Box-plot') #Titre du graphique
plt.ylim([0, max(data3)*1.20]) #
```

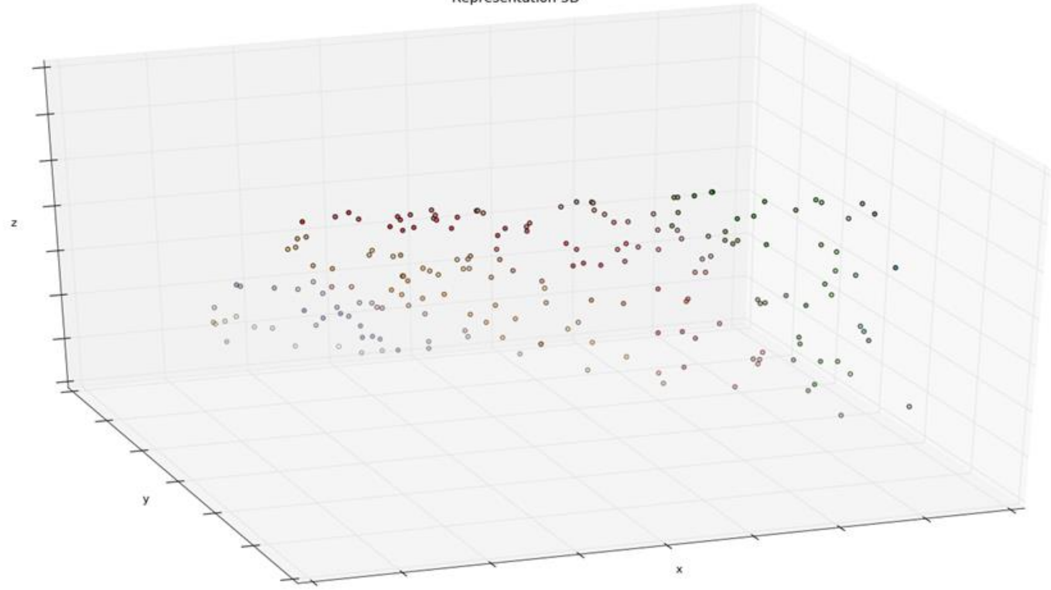
```
plt.show()
plt.cla()
plt.clf()
plt.close()
```



6.8. Les graphiques en 3D: exemple: Nuage de points en 3D

Soit la table de données suivante:

```
df_adv = pandas.read_csv('Advertising.csv', index_col=0)
x = df_adv[['TV']]
y = df_adv['Sales']
z = df_adv[['Radio']]
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
ax.scatter(x, y, z, c=y, cmap=plt.cm.Paired)
ax.set_title("Représentation 3D")
ax.set_xlabel("x")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("y")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("z")
ax.w_zaxis.set_ticklabels([])
fig.show()
fig.clear
```


Représentation 3D
z
y
x

Chapitre 7: Data mining et machine-learning sous python

Ce septième chapitre est consacré à la présentation des méthodes de data mining et des bases du marching-learning sous python. Les principales méthodes revues sont les méthodes de réduction de dimensions (ACP, AFC, ACM), les méthodes de classification (ACH, K-mean clustering, etc..), les méthodes de machine-learning(SVM, Random Forest, etc…). Pour la mise en application pratique de ces méthodes, nous nous servons du module sklearn-scikit.

7.1. Analyses en composantes principales (ACP)

Soit la table de données suivante:

```
import pandas
df=pandas.read_excel("iris.xlsx",sheetname="data",           header=0,
parse_cols="A:F")
print(df.head(n=5))
print(df.tail(n=5))
```

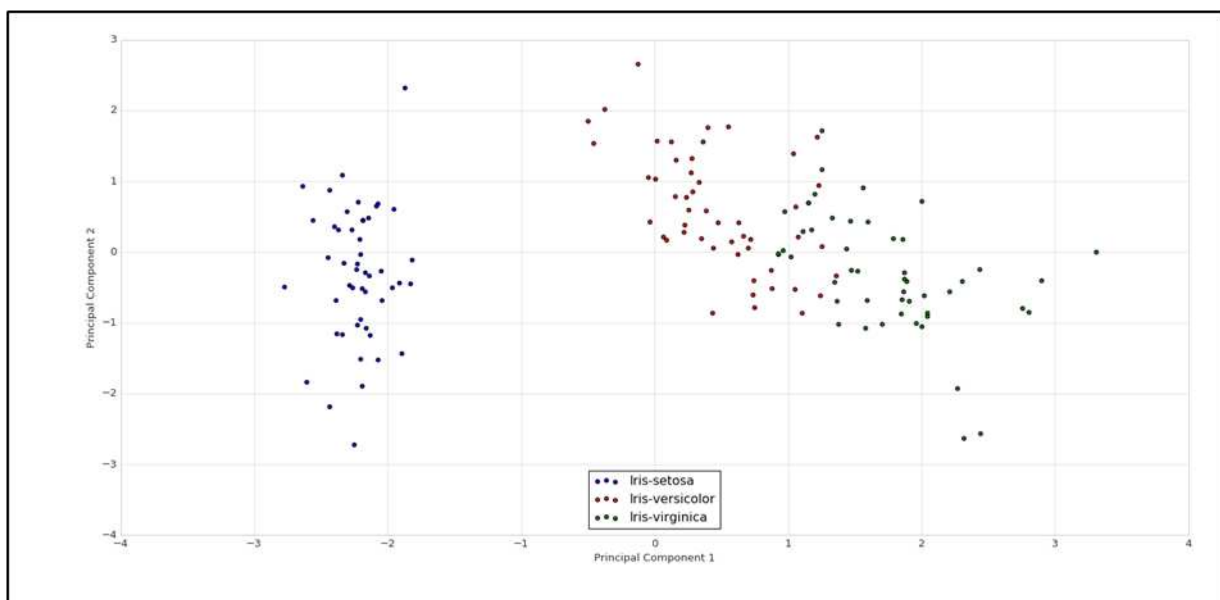
On souhaite faire l'ACP en considérant les variables quantitatives sepal_len sepal_wid petal_len et petal_wid. Il faut simplement signaler qu'il existe dans la base de données une variable catégorielle class qui permet de catégoriser les observations. Cette variable catégorielle a 3 valeurs (Iris-versicolor,Iris-virginica et Iris-setosa).

Avant toute chose, on va d'abord scinder cette table en deux sous tables: les variables quantitatives et la variable catégorielle.

NB : L'utilisation du module sklearn-scikit nécessite que les données soient en format verticale.

```
x = df.ix[:,0:4].values # Sélection des variables quantitatives et
conversion en forme verticale. On pouvait aussi utiliser
numpy.matrix()
y = df.ix[:,4].values # Sélection de la variable catégorielle et
conversion en format verticale.
### Ensuite, on va standardier les quatres variables de la table x
from sklearn.preprocessing import StandardScaler
x = StandardScaler().fit_transform(x) # on pouvait aussi utiliser
preprocessing.scale()
print(x)
######## Mise en oeuvre du PCA
from sklearn.decomposition import PCA as sklearnPCA
pca = sklearnPCA(n_components=2) # Retenir les deux composantes
score_data = pca.fit_transform(x) # prédiction des scores des deux
facteurs retenus
print('Variances: \n', pca.explained_variance_ratio_) # variances
expliquées
print(score_data) # scores prédits sur les deux composantes obtenues
```

```
# Affichage des variables sur les axes (corrélation des variables avec les axes)
print('Coordonnées des variables: \n', pca.components_ ) # il va falloir mettre en forme ces valeurs
col=df.iloc[:,0:4].columns.values # extraire les noms des colonnes à partir de la table iniiale.
print(pandas.DataFrame(pca.components_,columns=col, index=['composante1', 'composante2'])) # présenter sous forme de table.
### représentation des individus sur les deux axes (couleur en fonction des valeurs de la variable catégorielle)
import matplotlib.pyplot as plt
with plt.style.context('seaborn-whitegrid'):
    plt.figure(figsize=(6, 4))
    for lab, col in zip(('Iris-setosa', 'Iris-versicolor', 'Iris-virginica'),('blue', 'red', 'green')):
         plt.scatter(score_data[y==lab, 0], score_data[y==lab, 1],label=lab, c=col)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(loc='lower center')
plt.tight_layout()
plt.show()
plt.cla()
plt.clf()
plt.close()
```



7.2. Analyses factorielles (AFC)

Pour la mise en application de l'AFC, on va importer la base de données iris de python comme suit :

```
from sklearn.datasets import load_iris
iris = load_iris() # chargement de la table
x, y = iris.data, iris.target # extraire  les deux sous tables à partir de la table iris (data et target).
print(x) # base contenant quatre variables (ici continues)
print(y) # une seule variable (variable catégorielle)
######## mise en oeuvre  de l'AFC
from sklearn.decomposition import FactorAnalysis as FA
factor = FA(n_components=2,  random_state=101) # retenir les deux facteurs
score_data = factor.fit_transform(x) # prédiction des scores des deux facteurs retenus
print(score_data) # affiche les scores sur les deux facteurs
# Affichage des variables sur les facteurs (corrélation des variables avec les axes)
print('Coordonnées des variables: \n', factor.components_ ) # il va falloir mettre en forme ces valeurs.
#Mise en forme des valeurs
col=iris.feature_names # extraire les noms des colonnes.
#col=df.iloc[:,0:4].columns.values # extraire les noms des 4 premières colonnes pour le cas d'une table data frame ordinaire
print(pandas.DataFrame(factor.components_,columns=col,
index=['composante1', 'composante2'])) # présenter sous forme de table.
```

7.3. Analyses des correspondances multiples (ACM)

Par défaut, l'ACM se met en oeuvre lorsque la base données contient des variables quantiatives et catégorielles. Mais pour la mise en oeuvre pratique sous Python, il faut d'abord discrétiser toutes les variables quantitatives et ensuite transformer toutes les variables en variables binaires. On obtient alors le tableau disjonctif complet. C'est à partir de ce tableau que se met en oeuvre l'ACM (mca).

Toutefois, il faut noter que pour mettre en oeuvre l'ACM, il faut utiliser un module spécifique mca (qu'il faut installer au préalabale). Ainsi, on a:

```
# à revoir (erreur)
df=pandas.read_excel("iris.xlsx",sheetname="data",      header=0, parse_cols="A:F")
print(df.head(n=5))
print(df.tail(n=5))
import matplotlib.pyplot as plt
import mca
x=df.iloc[:,1:4]
mca_df=mca(x,benzecri=False)
```

```
#Valeurs singulières
print(mca_df.L)
### Composantes principales des colonnes (modalités)
print(mca_df.fs_c())
### Premier plan principal
col=[1,1,2,2,2,3,3,5,5,5,6,6,6,7,7,7]
plt.scatter(mca_df.fs_c()[:, 0],mca_df.fs_c()[:, 1],c=col)
```

7.4. Classification ascendante hiérarchique (CAH)

Pour la mise en œuvre de la classification CAH on va utiliser le module scipy.

Soit la table de données suivante:

```
fromage=pandas.read_excel("fromage.xlsx",sheetname="data",    header=0,
parse_cols="A:J")
print(fromage.head(n=5))
print(fromage.tail(n=5))
```

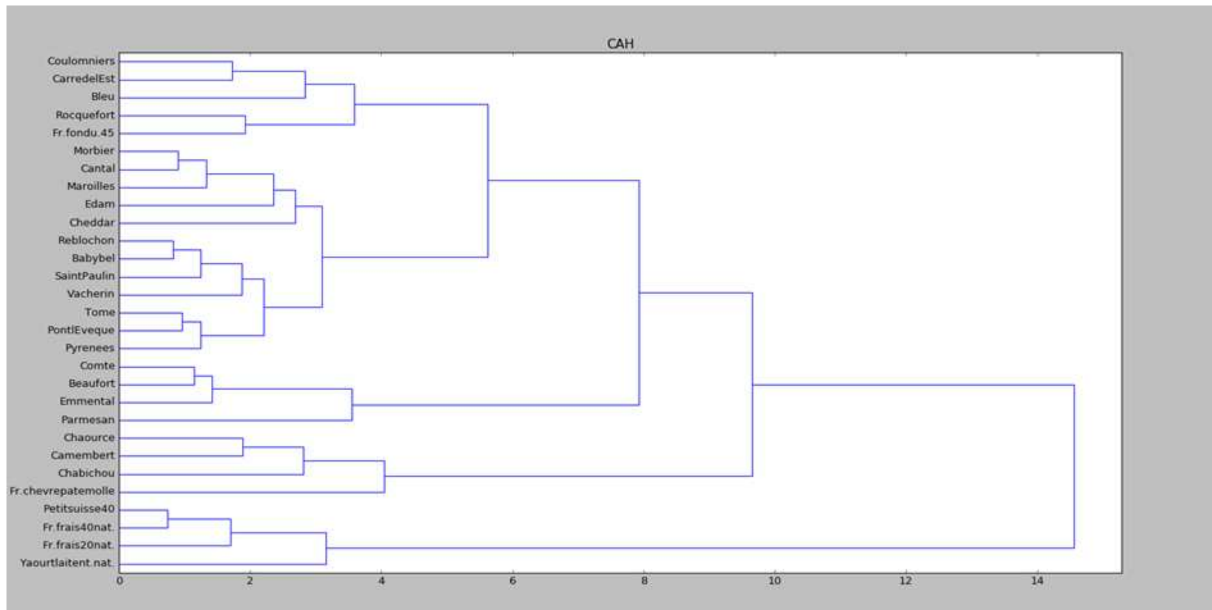
La base données contient un ensemble de fromages (29 observations) décrits par leurs propriétés nutritives (ex. protéines, lipides, etc. ; 9 variables). L'objectif est d'identifier des groupes de fromages homogènes, partageant des caractéristiques similaires.

La première variable nommée 'Fromages' représente les individus c'est à dire les types de fromage et les autres variables sont les variables actives.

Alors, on va transformer cette variable en index

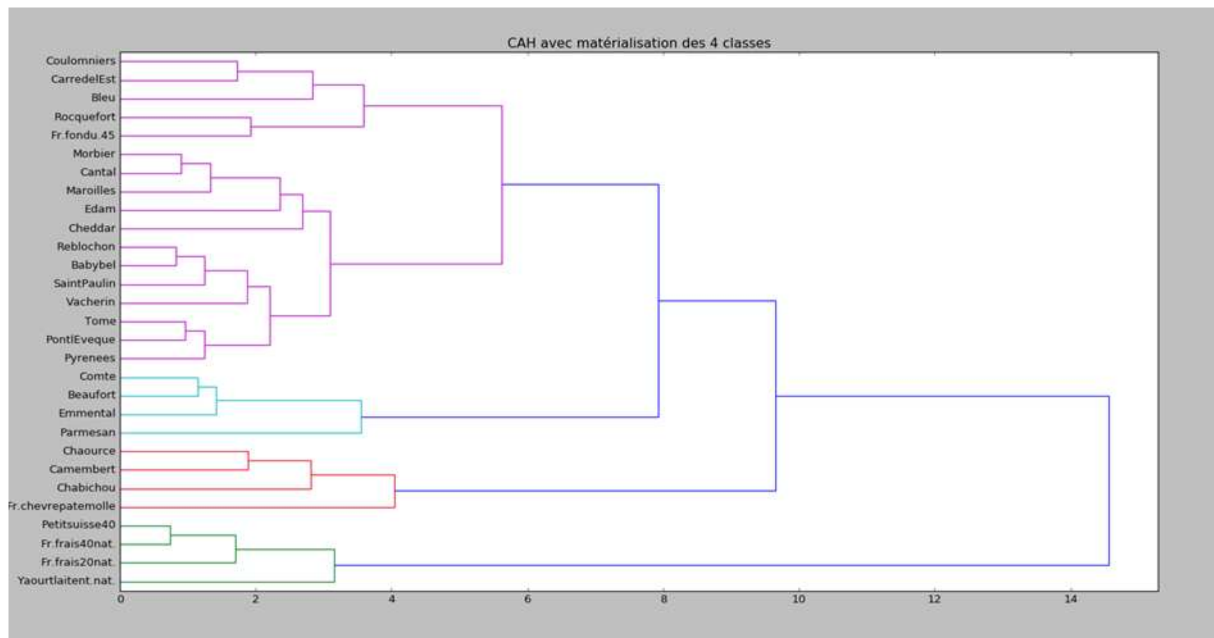
```
fromage = fromage.set_index('Fromages') # transformer la variable en
index
print(fromage.head(n=5))
#statistiques descriptives
print(fromage.describe())
#graphique - croisement deux à deux des variables
from pandas.tools.plotting import scatter_matrix
# scatter_matrix(fromage,figsize=(9,9)) # il ne doit pas y avoir de
variables caractères

######## Mise en œuvre de la CAH
from matplotlib import pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage #librairies
pour la CAH
#générer la matrice des liens
# il vaut lieux standardiser d'abord les variables:
from sklearn.preprocessing import StandardScaler
fromage_std = StandardScaler().fit_transform(fromage)
Z = linkage(fromage_std,method='ward',metric='euclidean')
#affichage du dendrogramme
plt.title("CAH")
```

```
dendrogram(Z,labels=fromage.index,orientation='left',color_threshold=0
,show_leaf_counts=True) # right, left bottom et top
plt.show()
```



Le dendrogramme « suggère » un découpage en 4 groupes. On ote qu'une classe de fromages, les « fromages frais » (tout à gauche), se démarque fortement des autres au point qu'on aurait pu envisager aussi un découpage en 2 groupes seulement. Nous y reviendrons plus longuement lorsque nous mixerons la CHA avec une analyse en composantes principales (ACP). Ainsi au vu du graphique,on va retenir 4 classes en coupant l'arbre au niveau de 7

```
plt.title('CAH avec matérialisation des 4 classes')
dendrogram(Z,labels=fromage.index,orientation='left',color_threshold=7
,show_leaf_counts=True) #matérialisation des 4 classes (hauteur t = 7)
avec les couleurs
plt.show()
plt.cla()
plt.clf()
plt.close()
```




```
#découpage à la hauteur t = 7 ==> identifiants de 4 groupes obtenus
import scipy.cluster.hierarchy as cah
groupes_cah = cah.fcluster(Z,t=7,criterion='distance')
print(groupes_cah)
#index triés des groupes
import numpy as np
idg = np.argsort(groupes_cah)
#affichage des observations et leurs groupes
cluster_data=pandas.DataFrame(groupes_cah[idg], fromage.index[idg]) # données avec cluster
# transformer l'index en variable:
cluster_data.reset_index(level=0, inplace=True)
cluster_data=cluster_data.rename(columns={0: 'cluster'}, inplace=False) # renommer la colonne 0 comme cluster (car par défaut c'est 0 comme nom de la colonne)
print(cluster_data)# Ces données étant crées, on peut ensuite faire merge avec la table intiale pour avoir la base complète.
# Ensuite, on peut faire l'ACP sur les variables et représenter les individus sur les deux axes factoriels (en distinguant les clusters par les couleurs)
```

7.5. K-Means clustering (méthodes des centroides)

Cette méthode de classification fait partie de la méthode générale de classification non supervisée. Pour sa mise en œuvre sous python, on va utiliser le module sklearn-scitkit.

Soit la table de fromage (précédemment utilisée pour la classification cah) :

```
fromage=pandas.read_excel("fromage.xlsx",sheetname="data", header=0, parse_cols="A:J")
print(fromage.head(n=5))
```

```
print(fromage.tail(n=5))
# La variable nommée 'Fromages' représente les individus, on va transformer cette variable en index
fromage = fromage.set_index('Fromages') # transformer la variable en index
print(fromage.head(n=5))
#statistiques descriptives
print(fromage.describe())
#from matplotlib import pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage #librairies pour la CAH
# standardiser d'abord les variables:
from sklearn.preprocessing import StandardScaler
fromage_std = StandardScaler().fit_transform(fromage)
```

La méthode K-means clsutering sera mise en œuvre sur cette table standardiée.

NB : Bien qu'on utilize le module sklearn pour réaliser le k-means, la table fromage_std n'a pas besoin d'être mise en format verticale (un exception avec le module sklearn).

Par ailleurs, il faut noter qu'il y a deux manières de mettre en œuvre la méthode de K-mean clustering en fonction du choix des centroïdes. On distingue les centroïdes fixe et les centroïdes mobiles

7.5.1. Méthodes des centroides fixes (fixer le nombre de clusters)

```
# Mise en en oeuvre
import numpy as np
from sklearn import cluster
kmeans = cluster.KMeans(n_clusters=4) # Nombre de clusters fixé à 4
kmeans.fit(fromage_std) # la numérotation des clusters commence à 0
idk = np.argsort(kmeans.labels_) #index triés des clusters
cluster_data=pandas.DataFrame(kmeans.labels_[idk],fromage.index[idk])
#affichage des observations et leurs groupes
print(cluster_data)
#distances aux centres de classes des observations
print(kmeans.transform(fromage_std)) # Distances aux centres de classes pour chaque individu ( les min. respectifs correspondent à la classe)
#Comparaisons avec avec les groupes obtenus avec la méthode CAH précédente.
print(pandas.crosstab(groupes_cah,kmeans.labels_))
```

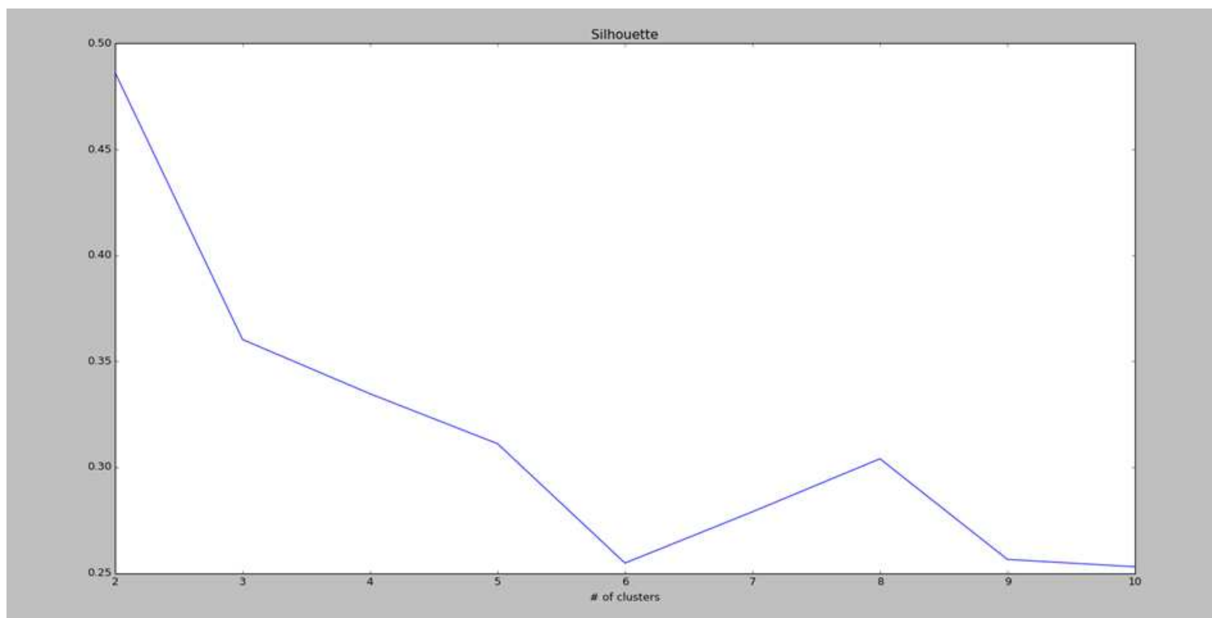
7.5.2. Méthodes des centroides mobiles (aide à la détection du nombre adéquat de clusters)

La méthode de K-MEANS, à la différence de la CAH, ne fournit pas d'outils d'aide à la détection du nombre de clusters. Nous devons les programmer sous Python ou utiliser des procédures

proposées par des packages dédiés. La démarche générale est la suivante : on fait varier le nombre de groupes et on surveille l'évolution d'un indicateur de qualité de la solution c.-à-d. l'aptitude des individus à être plus proches de ses congénères du même groupe que des individus des autres groupes.

Par exemple, dans ce qui suit, on calcule la métrique « silhouette » pour différents nombres de groupes issus de la méthode des centres mobiles.

```
from sklearn import metrics #librairie pour évaluation des partitions
#faire varier le nombre de clusters de 2 à 10 #utilisation de la métrique "silhouette"
import numpy as np
res = np.arange(9,dtype="double")
for k in np.arange(9):
 km = cluster.KMeans(n_clusters=k+2)
 km.fit(fromage_std) # fromage_std est la table de donnée centrée-reduite
 res[k] = metrics.silhouette_score(fromage_std,km.labels_)
print(res)
#graphique
import matplotlib.pyplot as plt
import numpy as np
plt.title("Silhouette")
plt.xlabel("# of clusters")
plt.plot(np.arange(2,11,1),res)
plt.show()
plt.cla()
plt.clf()
plt.close()
```



La partition en k = 2 groupes semble la meilleure au sens de la métrique « silhouette » la plus élevées

Exemple d'application: clustring d'une image (attention temps d'execution trop long !)

```
### On va utiliser le module misc de scipy pour importer l'image nommée face.
# imporation:
import scipy.misc
face = scipy.misc.face()
face.shape
face.dtype
# représentation
import matplotlib.pyplot as plt
plt.gray()
plt.imshow(face)
plt.show()
plt.cla()
plt.clf()
plt.close()
#Appliqons le k-means clustering sur les observations qui constituent cette image en prenant k=5
from scipy import misc
face = misc.face(gray=True).astype(np.float32)
X = face.reshape((-1, 1))  # We need an (n_sample, n_feature) array
K = k_means = cluster.KMeans(n_clusters=5)  # 5 clusters
k_means.fit(X)
values = k_means.cluster_centers_.squeeze()
labels = k_means.labels_
face_compressed = numpy.choose(labels, values)
face_compressed.shape = face.shape
```

7.6. Aide à l'interprétation des clusters (projection des clusters sur des axes factoriels issus d'une ACP)

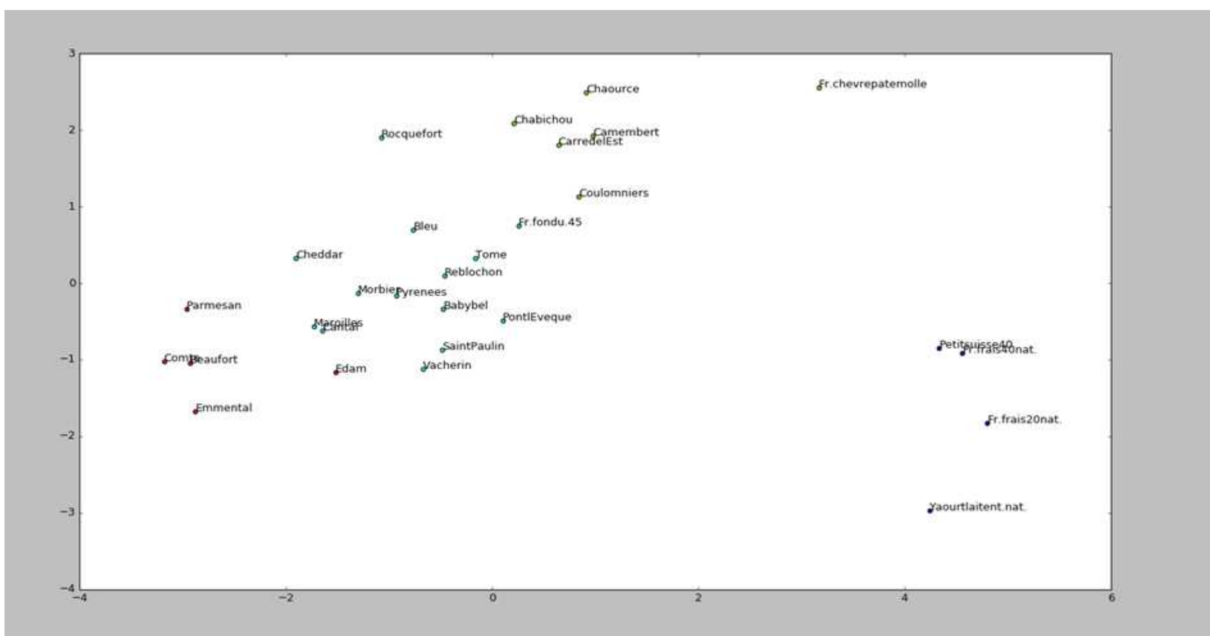
Le but ici est de représenter les individus sur les axes factotiels en les distinguant selon les clusters (couleurs différentes) afin d'aider à interpréter les clusters.

Soit fromage_std lat table standardisée des variables de la table fromage construite dans les sections CAH et K-means(voir plus haut pour les étapes de construction). Pour la projection des clusters sur les axes factotoriels, on suit deux étapes :

Première étape : Réalisation de l'ACP

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
acp = PCA(n_components=2).fit_transform(fromage_std) # les scores les deux composantes retenues
```

```
#projeter dans le plan factoriel avec un code couleur différent selon
le groupe
for  couleur,k  in  zip(['red','blue','lawngreen','aqua'],[0,1,2,3]):  #
on associe une couleur à chaque cluster en utilisant la fonction zip

plt.scatter(acp[kmeans.labels_==k,0],acp[kmeans.labels_==k,1],c=couleu
r) # le 0 et le 1 signifie première et seconde compodantes.
#mettre les labels des points (remarquer le rôle de enumerate())
for i,label in enumerate(fromage.index):
    plt.annotate(label,(acp[i,0],acp[i,1]))
 plt.show()
plt.cla()
plt.clf()
plt.close()
```

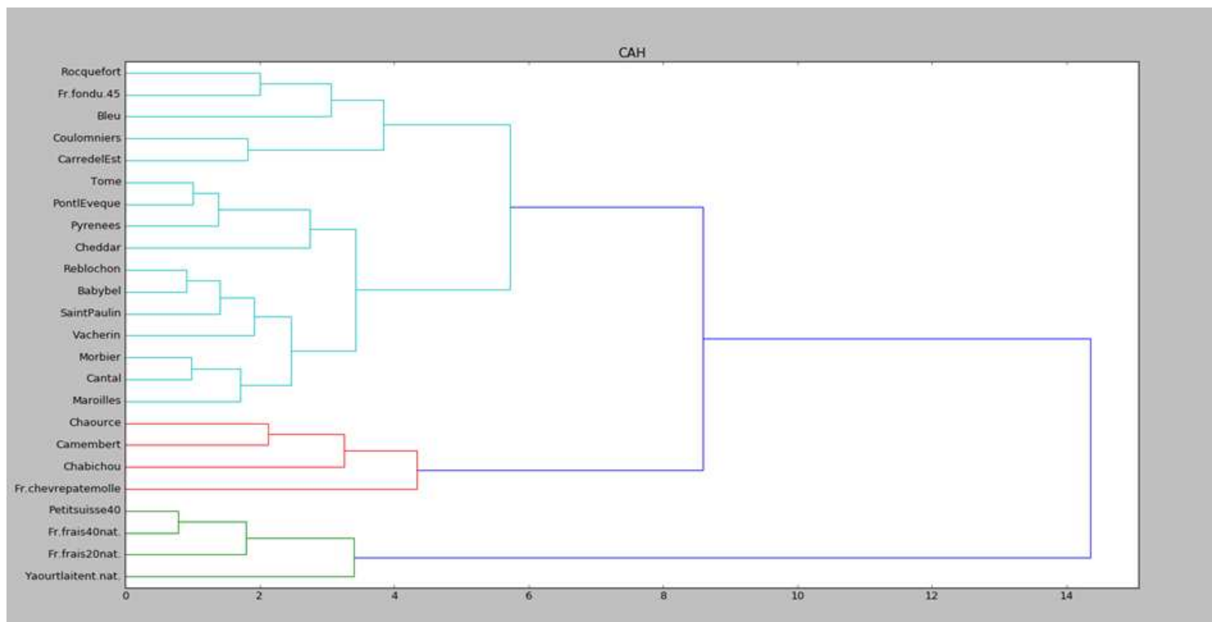


On remarque un léger souci. Le groupe des fromages frais (n° de groupe = 0) écrase l'information disponible et tasse les autres fromages dans un bloc qui s'oriente différemment. De fait, si l'on comprend bien la nature du groupe n°0 des fromages frais, les autres sont plus compliqués à comprendre lorsqu'ils sont replacés dans le premier plan factoriel.

Pour raffiner l'analyse, on va faire quelques traitements. Retirer les fromages frais du jeu de données. Les fromages frais sont en quelque sort atypiques – éloignés de l'ensemble des autres observations – qu'ils masquent des relations intéressantes qui peuvent exister entre ces produits. Nous reprenons l'analyse en les excluant des traitements.

```
#retirer des observations le groupe n°0 du k-means précédent
fromage_subset = fromage.iloc[kmeans.labels_!=0,:]
print(fromage_subset.shape)
#centrer et réduire
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing
```

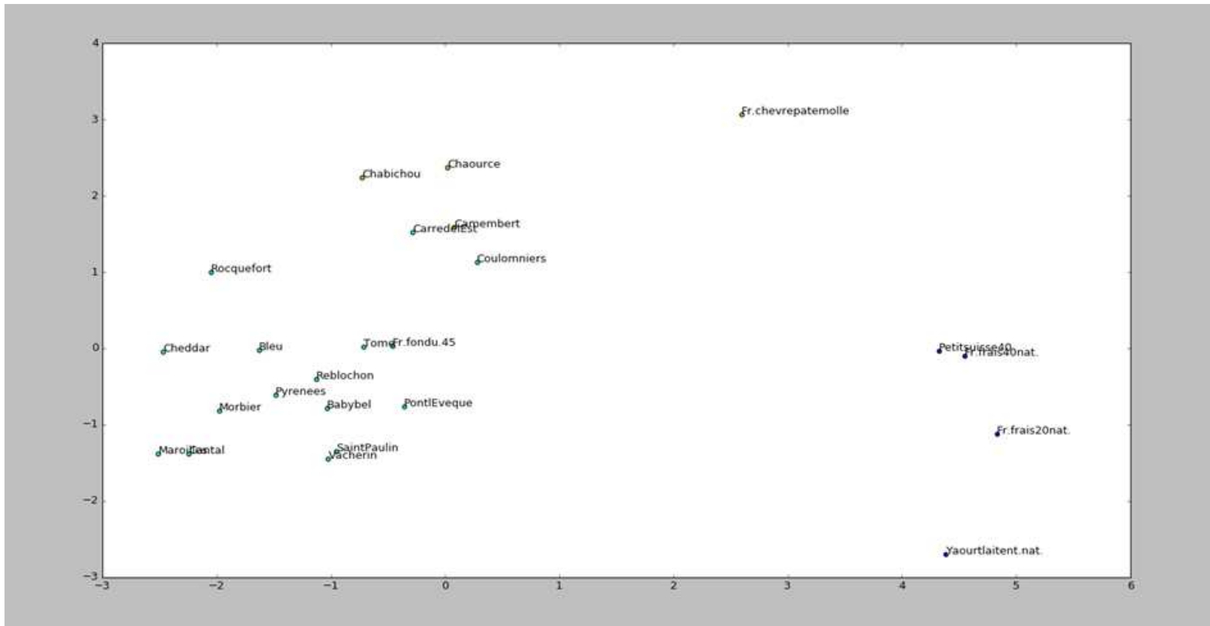
```
fromage_subset_cr = preprocessing.scale(fromage_subset)
#générer la matrice des liens
Z_subset = linkage(fromage_subset_cr,method='ward',metric='euclidean')
#cah et affichage du dendrogramme
plt.title("CAH")
dendrogram(Z_subset,labels=fromage_subset.index,orientation='left',col
or_threshold=7)
plt.show()
plt.cla()
plt.clf()
plt.close()
```



3 groupes se distinguent. Et le phénomène d'écrasement constaté dans l'analyse précédente est amoindri.

```
#groupes
import scipy.cluster.hierarchy as cah
groupes_subset_cah = cah.fcluster(Z_subset,t=7,criterion='distance')
print(groupes_subset_cah)
#ACP
acp_subset = PCA(n_components=2).fit_transform(fromage_subset_cr)
#projeter dans le plan factoriel #avec un code couleur selon le groupe
plt.figure(figsize=(10,10))
for couleur,k in zip(['blue','lawngreen','aqua'],[1,2,3]):

plt.scatter(acp_subset[groupes_subset_cah==k,0],acp_subset[groupes_sub
set_cah==k,1],c=couleur)
#mettre les labels des points (remarquer le rôle de enumerate())
for i,label in enumerate(fromage_subset.index):
    plt.annotate(label,(acp_subset[i,0],acp_subset[i,1]))
plt.show()
plt.cla()
```

```
plt.clf()
plt.close()
```



7.7. Classification par les méthodes Support Vector Machine (SVM)

Le but de la méthode SVM pour la classification est d'utiliser un échantillon d'apprentissage pour calculer des critères de classification et éventuellement appliquer ces critères sur un nouveau échantillon afin de classer les individus qu'il contient (prédictions). On a donc un échantillon de test (apprentissage) et un échantillon de prédiction. Il faut noter que l'échantillon de test contient toujours une variable catégorielle qui détermine la classification réelle des individus. Le but du SVM est d'utiliser l'échantillon de prédiction pour générer une variable catégorielle permettent de classifier les individus de l'échantillon de prédiction.

Pour appliquer cette méthode, on va considérer la base de données iris (comme échantillon d'apprentissage). On a :

```
df=pandas.read_excel("iris.xlsx",sheetname="data",           header=0,
parse_cols="A:F")
print(df.head(n=5))
print(df.tail(n=5))
```

Cette base de données contient trois catégories de fleurs (Iris setosa, Iris versicolor, Iris virginica) observé sur quatre variables: sepal_length, sepal_width, petal_length and petal_width

Phase Learning

Nous allons d'abord faire du learning avant de faire du prédicting. Pour cela, nous allons utiliser le module SVM (Support Vector Machine). Mais avant, nous allons scinder la table en deux parties. La partie données et la partie catégorie:

```
dfv=numpy.matrix(df) # mise de la table sous la forme verticale (format numpy array)
data=dfv[:,0:4] # extraction des données
target=dfv[:,4:] # extraction des catégories
```

Pour faire du learning, nous allons utilizer la function SVC qui est l'algorithme SVM à but de classification.

```
#mise en oeuvre
# learning
from sklearn import svm
clf = svm.SVC(kernel='linear') # définition de l'estimateur ( kernel définit le type d'algorithe. On distingue 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable)
```

En effet, les classes ne sont pas toujours séparées par un hyperplan (c'est à dire droite linéaire ex: mélange eau et huile)). Elles peuvent être séparées de manière polynomiale (séparée par des lignes non uniforme: ex: mélange de deux liquides de même lourdeur). Les classes peuvent avoir des séparations radiales RBF (Radial Basis Function). Ex: des gouttes d'huile qui flottent un peu partout à la surface de l'eau mais pas de manière regroupée. Des petites gouttes étalées par ci par là.

```
#Estimation
clf.fit(data, target) # apprentissage (estimation du modèle)
print(clf.coef_  ) # matrice des coefficients
print(clf.intercept_) # La constante
print(clf.support_ ) # Indices of support vectors.
print(clf.support_vectors_ ) # Support vectors.
print(clf.support_ ) # n_support_
print(clf.dual_coef_ ) #
print(clf.intercept_  ) #
```

Phase de prédiction

Une fois qu'on a estimé le modèle, on peut maintenant faire des prévisions pour déterminer la catégorie à laquelle appartient les données(de l'échantillon de prédiction).

Par exemple: soit les données suivantes 5.5,3.5,1.3,0.2 qui correspondent aux valeurs hors échantillon de sepal_length, sepal_width, petal_length and petal_width (on choisit ici un echantillon à 1 individus). On pouvait aussi choisir plusieurs spécifiée sous forme d'array (voir plus bas).

On veut déterminer à quelle catégorie appartient cette observation. Il faut savoir que les catégories [['Iris setosa', 'Iris versicolor', 'Iris virginica']] correspondent à un array dont les

indices [[0, 1, 2]]. Dès lors il s'agit de déterminer le libellé ou le numéros de l'index correspondant pour l'observation. Ainsi, on a:

```
x_pred=[[5.5,3.5,1.3,0.2 ]] # définition du point de prédiction
clf.predict(x_pred) # le résulat est ['Iris-setosa'] array([0])
print(clf.predict(x_pred)) # donne ['Iris-setosa']
```

Pour faire la prédiction sur plusieurs observations de l'échantillon de test, on élabor une boucle: Exemple: Soit x_pred un echantillon sous format vertical (array). La prédiction sur plusieurs individus se fait comme suit :

```
for i in range(len(x_pred)):
    print(clf.predict(x_pred[i]))
```

D'une manière générale, pour appliquer cette méthode SVM.SVC, on scinde l'échantillon en deux: échantillon d'apprentissage et un échantillon de test.

Remarque: Il faut noter que les regressions logisqtiques binaires qui sont des cas particuliers de classifications peuvent aussi être réalisées en utilisant les méthodes de machine_learning comme SVM qui classent les individus entre les deux groupes.

Par ailleurs, il faut signaler qu'il existe plusieurs autres algorithme SVM. Certains à but de classification (comme svm.SVC, svm.NuSVC et svm.LinearSVC) et d'autres à but de regression de type SVM (comme svm.SVR). Consulter la documentation sur SVM.

7.8. Classification par les méthodes de Random Forest

Le principe de la méthode de Random Forest est d'utiliser un grand nombre d'arbres de décision construits chacun avec un sous-échantillon différent de l'échantillon d'apprentissage, et pour chaque construction d'arbre, la décision à un noeud est fait en fonction d'un sous-ensemble de variables tirées au hasard. Puis, on utilise l'ensemble des arbres de décision produits pour faire la prédiction, avec un vote à la majorité (pour la classification avec variable prédite de type facteur) ou une moyenne (pour de la régression, avec variable prédite de type numérique).

La démarche générale de mise en œuvre est la suivante :

1-Boostrap: on construit K arbres à partir de sous échantillons de Dn tirés avec remise.

2-Sélection des variables de façon aléatoire pour construire l'arbre de décision.

3-Moyennage des arbres obtenus et élagage de l'arbre

Pour la mise en œuvre pratique du Random forest, on va considérer la base de données iris (comme échantillon d'apprentissage)

```
df=pandas.read_excel("iris.xlsx",sheetname="data",          header=0,
parse_cols="A:F")
print(df.head(n=5))
print(df.tail(n=5))
```

Cette base de données contient trois catégories de fleurs (Iris setosa, Iris versicolor, Iris virginica) observé sur quatre variables: sepal_length, sepal_width, petal_length and petal_width

Nous allons d'abord faire du learning. Pour cela, nous allons utiliser le module sklearn.ensemble. Mais avant, nous allons scinder la table en deux partie. La partie données et la partie catégorie:

```
dfv=numpy.matrix(df) # mise de la table sous la forme verticale (numpy
array)
data=dfv[:,0:4] # extraction des données
target=dfv[:,4:] # extraction des catégories
#target=df['class'] # extraction des catégories
# Mise en oeuvre
# Learning
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(n_estimators=10,            criterion='gini',
bootstrap=True, random_state=100) # Définition de l'estimateur
rf.fit(data, target) # apprentissage (estimation du modèle)
# prédiction
#  soit  x  un  point  dont  les  valeurs  sur  les  quatres  variables
explicatives sont: [5.5,3.5,1.3,0.2 ]]. on a:
x_pred=[[5.8,2.7,4.1,1 ]] # définition du point de prédiction
print(rf.predict(x_pred))  #  Déterminer  la  catégories  (target)  à
laquelle appartient le point indiqué.
```

Pour faire la prédiction sur plusieurs observations de l'échantillon de test, on élabor une boucle. Exemple: Soit x_pred un echantillon sous format vertical (array). La prédiction sur plusieurs individus se fait comme suit :

```
for i in range(len(x_pred)):
    print(rf.predict(x_pred[i]))
```

7.9. Classification par les méthode des voisins les plus proches

Le but des méthode des voisins les plus proches est de déterminer des critères de ressemblances entre les observations afin des les regrouper dans des classes homogènes. Dans cette section, nous allons présenter deux approches: les k-voisins les plus proches et la méthode des voisins dans un rayon (radius)

On va considérer la base de données iris (comme échantillon d'apprentissage)

```
df=pandas.read_excel("iris.xlsx",sheetname="data",           header=0,
parse_cols="A:F")
print(df.head(n=5))
print(df.tail(n=5))
```

Cette base de données contient trois catégories de fleurs (Iris setosa, Iris versicolor, Iris virginica) observé sur quatre variables: sepal_length, sepal_width, petal_length and petal_width. Mais avant, nous allons scinder la table en deux partie. La partie données et la partie catégorie:

```
dfv=numpy.matrix(df) # mise de la table sous la forme verticale (numpy
array)
data=dfv[:,0:4] # extraction des données
target=dfv[:,4:] # extraction des catégories
```

7.9.1. Méthode des k-voisins les plus proches

```
#mise en oeuvre
# Learning
from sklearn.neighbors import NearestNeighbors
knn = NearestNeighbors(n_neighbors=5, radius=1.0, algorithm='auto',
leaf_size=30, metric='minkowski', p=2) # définition de l'estimateur
knn.fit(data, target) # apprentissage (estimation du modèle)
## Prediction:
# soit x un point dont les valeurs sur les quatres variables
explicatives sont: [5.5,3.5,1.3,0.2 ]]. on a:
x_pred=[[5.5,3.5,1.3,0.2 ]] # définition du point de prédiction
print(knn.kneighbors(x_pred, 5, return_distance=False)) # Renvoie la
liste des indices des 5 individus (dans la base de départ) qui sont
les proches du point indiqué. Pour voir les distance on fait
return_distance=True
print(knn.kneighbors_graph(x_pred,                    n_neighbors=5,
mode='connectivity')) # représentation des 5 voisins les plus proches
avec pondération
print(knn.radius_neighbors(x_pred, radius=0.3, return_distance=False))
# Renvoie la liste des indices des individus (dans un rayon de 0.3)
autour du point indiqué. Pour voir les distance on fait
return_distance=True
```

Pour faire la prédiction sur plusieurs observations de l'échantillon de test, on élabor une boucle. Exemple: Soit x_pred un echantillon sous format vertical (array). La prédiction sur plusieurs individus se fait comme suit :

```
for i in range(len(x_pred)):
    print(knn.kneighbors(x_pred[i]))
    print(knn.kneighbors_graph(x_pred[i]))
    print(knn.radius_neighbors(x_pred[i]))
```

7.9.2. Méthode des voisins proches dans un rayons

```
## Learning:
from sklearn.neighbors import RadiusNeighborsClassifier
rnn = RadiusNeighborsClassifier(radius=0.3,weights='distance', algorithm='auto', leaf_size=30, metric='minkowski', p=2) # définition de l'estimateur
rnn.fit(data, target) # apprentissage (estimation du modèle)
## Prediction:
# soit x un point dont les valeurs sur les quatres variables explicatives sont: [5.5,3.5,1.3,0.2 ]]. on a:
x_pred=[[5.8,2.7,4.1,1 ]] # définition du point de prédiction
print(rnn.predict(x_pred)) # Déterminer la catégories (target) à laquelle appartient le point indiqué
print(rnn.radius_neighbors(x_pred, radius=0.2, return_distance=False)) # Renvoie la liste des indices des individus (dans un rayon de 0.2) autour du point indiqué. Pour voir les distance on fait return_distance=True
# Pour faire la prédiction sur plusieurs observations de l'échantillon de test, on élabor une boucle: Ex:
for i in range(len(x_pred)):
    print(rnn.predict(x_pred[i]))
    print(rnn.radius_neighbors(x_pred[i]))
```

Remarque: il faut noter que la méthode des k-voisins les plus proches est une méthode non supervisées (Unsupervised Nearest Neighbors)

Par ailleurs, il faut noter qu'on peut aussi utiliser les méthodes de 'Nearest Neighbors Regression' lorsque la variable qui représente les catégores est quantitatives. On a deux principales méthodes: KNeighborsRegressor et RadiusNeighborsRegressor (voir documentation scikit)

7.10. Recherche des valeurs optimales des paramètres dans les algorithmes (GRID SEARCH)

De nombreux algorithmes de machine learning reposent sur des paramètres qui ne sont pas toujours évidents à déterminer pour obtenir les meilleurs performances sur un jeu de données à traiter. Les lagorithmes dépendent donc des paramètres dont les valeurs doivent être fixées par l’utilisateur. D'où la nécessité de trouver des valeurs optimales.

Exemple : Nous allons utiliser la base de données diabetes pour prédire la probabalité du diabètes en fonction des antécédents cliniques. Pour cela nous allons partir de l'algorithme svm.SVC.

Soit la table de données

```
df=pandas.read_excel("diabetes.xlsx",sheetname="data",        header=0,
parse_cols="A:I")
print(df.head(n=5))
# Y : variable cible (diabète)
#transformation en matrice numpy (format verticale) - seul reconnu par
scikit-learn
data = df.as_matrix() #
# data=numpy.matrix(df) # seconde méthode
X = data[:,0:8] # X matrice des varibale explicatives les 8 première
variables
y = data[:,8] #y vecteur de la variable à prédire
#utilisation du module cross_validation de scikit-learn (sklearn) pour
subdiviser l'échantillon en un égantillon d'apprentissage et de
validation
from sklearn import cross_validation
#subdivision des données – éch.test = 300 ; éch.app = 768 – éch.test =
468
X_app,X_test,y_app,y_test                                              =
cross_validation.train_test_split(X,y,test_size = 300,random_state=0)
print(X_app.shape,X_test.shape,y_app.shape,y_test.shape)
#Application svm
from sklearn import svm
#par défaut un noyau RBF et C = 1.0
mvs = svm.SVC()
#modélisation
modele2 = mvs.fit(X_app,y_app)
#prédiction echantillon test
y_pred2 = modele2.predict(X_test)
#matrice de confusion
print(metrics.confusion_matrix(y_test,y_pred2))
#succès en test
print(metrics.accuracy_score(y_test,y_pred2)) # 0.67
```

On constate que la méthode ne fait pas mieux que le classifieur par défaut (prédire systématiquement 'negative', classe majoritaire). Matrice de confusion

La question alors est de savoir si c'est la méthode (SVM) qui est inapte ou c'est le paramétrage qui est inadapté ? D'où la nécessité de faire un paramétrage optimale.

```
#Détermination des meilleurs valeurs des paramètres
### Stratégie : Grille de recherche. On indique les paramètres à faire
varier, scikit-learn les croise et mesure les performances en
validation croisée.
###import de la classe
from sklearn.grid_search import GridSearchCV
###combinaisons de paramètres à évaluer
parametres = [{'C':[0.1,1,10],'kernel':['rbf','linear']}]
```

```
###évaluation en validation croisée de 3 x 2 = 6 configurations. accuracy sera le critère à utiliser pour sélectionner la meilleure config
### mvs est l'instance de la classe svm.SVC
grid = GridSearchCV(estimator=mvs,param_grid=parametres,scoring='accuracy')
###lancer la recherche – attention, long en calculs
grille = grid.fit(X_app,y_app)
###résultat pour chaque combinaison
print(grille.grid_scores_)
###meilleur paramétrage
print(grid.best_params_) # {'C' : 10, 'kernel' : 'linear'}
###meilleur performance – estimée en interne par validation croisée
print(grid.best_score_) # 0.7564
###prédiction avec le modèle « optimal » c.-à-d. {'C' : 10, 'kernel' : 'linear'}
y_pred3 = grille.predict(X_test)
###taux de succès en test
print(metrics.accuracy_score(y_test,y_pred3)) # 0.7833, on se rapproche de la rég. logistique
```

Annexe : Exercices et cas pratiques de programmation (résolus)

Exercices de programmation de base, de gestions de fchiers et de traitement de textes

Exercice 1 : Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7.

Résolution

```
# défintion de la fonction multipli7
def multipli7():
    n=1
    while n<=20:
        print("7","x",n, "=",7*n, sep=" ")
        n=n+1
# appel de la fonction
multipli7()
```

Exercice 2 : Écrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros vers le dollar canadien. La progression des sommes de la table sera « géométrique », comme dans l'exemple ci-dessous : 1 euro(s) = 1.65 dollar(s) ; 2 euro(s) = 3.30 dollar(s) ; 4 euro(s) = 6.60 dollar(s) ; 8 euro(s) = 13.20 dollar(s), etc. (S'arrêter à 16384 euros.)

Résolution

D'abord, on remarque que les valeurs de la table suivent une progression géométrique de raison q=2 et de premier terme u1=1.65

Pour écire le programme, on a alors deux choix. Soit on écrit la formule générale de la suite géométrique en utilisant q et u1 ou bien on utilise la formule réccurente Un+1=qxUn.

```
# première méthode: formule générale: u1*((q)**n
def conversion1():
    n=0
    u1=1.65
    q=2
    val=u1*((q)**n)
    while val<16384:
        print(n+1, "euro(s)=",val, "dollar(s)", sep=" ")
        n=n+1
        val=u1*((q)**n)
# appel
conversion1()
# deuxième méthode: formule de récurence Un+1=Unxq
def conversion2():
    n=0
    u1=1.65
    q=2
    val=u1*((q)**n)
    while val<16384:
        print(n+1, "euro(s)=",val, "dollar(s)", sep=" ")
        n=n+1
        val=val*q
# appel
conversion2()
# on voit que les deux méthodes donnent le même résultat.
```

Exercice 3 : Écrivez un programme qui calcule le volume d'un parallélépipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.

Résolution

```
# on sait que le volume d'un parallélépipède est égale Longueur x Largeur x Hauteur (LxlxH)
# définition de la fonction
def volparal():
    try:
        L=float(input("Donner la longueur:"))
        l=float(input("Donner la largeur:"))
        H=float(input("Donner la hauteur:"))
    except ValueError:
```

```
        print('Vous devez entrer un nombre')
        L=float(input("Donner la longueur:"))
        l=float(input("Donner la largeur:"))
        H=float(input("Donner la hauteur:"))
    print("le volume calculé est",L*l*H, sep=' ')
# appel
volparal()
# définition des exceptions à revoir
```

Exercice 4 : Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3. Exemple : 7 14 21 * 28 35 42 * 49 ...

```
# Résolution
# défintion de la fonction multipli7
def multiple7Star():
    n=1
    while n<=20:
        if 7*n%3==0:
            print("7","x",n, "=",7*n, "*")
        else:
            print("7","x",n, "=",7*n)
        n=n+1
# appel de la fonction
multiple7Star()
```

Exercice 5 : Écrivez un programme qui affiche la suite de symboles suivante :

```
*
**
***
****
*****
******
*******
```

Résolution:

```
# Idée: on peut fixer un compteur de 1 à 10. Et pour chaque valeur, on fait une boucle pour faire print * , en faire une liste et transformer cette liste en caractère unique.
def printStar():
    i=1
    while i<=30:
```

```
        listStar=[]
        for s in range(1,i+1):
            listStar.append("*")
            chStar= "".join(listStar)
        print(chStar)
        i=i+1
printStar()
```

Exercice 6 : Écrivez un programme qui calcule les intérêts accumulés chaque année pendant 20 ans, par capitalisation d'une somme de 100 euros placée en banque au taux fixe de 4,3 %

Résolution :

```
def calculInteret():
     v0=100
     ti=0.043
     n=1
     v=v0*(1+ti)
     I=ti*v0
     IC=I
     while n<=20:
         print("la valeur après",n, "an(s) est",v)
         print("l'intérêt après",n, "an(s) est",I)
         print("l'intérêt cumulé après",n, "an(s) est",IC)
         v=v*(1+ti)
         I=v*ti
         IC=IC+I
         n=n+1
calculInteret()
```

Exercice 7 : Soient les listes suivantes :

t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']

Écrivez un petit programme qui crée une nouvelle liste t3. Celle-ci devra contenir tous les éléments des deux listes en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant : ['Janvier',31,'Février',28,'Mars',31, etc...].

Résolution :

```
# Remarque: chacune des deux listes a un nombre d'élement identique. On peut alors récupérer les éléments par leur index et créer une nouvelle liste.
def newlist():
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
    n=0
    t3=[]
    while n<=11:
        t3.append(t2[n])
        t3.append(t1[n])
        n=n+1
    print(t3)
newlist()
```

Exercice 8 : Écrivez un programme qui recherche le plus grand élément présent dans une liste donnée. Par exemple, si on l'appliquait à la liste [32, 5, 12, 8, 3, 75, 2, 15], ce programme devrait afficher : le plus grand élément de cette liste a la valeur 75.

Résolution

```
# on peut utiliser la fonction reduce
def maxval():
    vallist=[32, 5, 12, 8, 3, 75, 2, 15]
    from functools import reduce
    print("la valeur maximum est ",reduce(max, vallist))
    maxval()
```

Exercice 9 : Écrivez un programme qui analyse un par un tous les éléments d'une liste de mots (par exemple : ['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']) pour générer deux nouvelles listes. L'une contiendra les mots comportant moins de 6 caractères, l'autre les mots comportant 6 caractères ou davantage.

Résolution

```
# On va utiliser une boucle for (on pouvait aussi récupérer chaque élement avec son index et compter le nombre de caractères. Ensuite définir une clause if:)
def nbChar():
    listChar=['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']
    list6g=[]
```

```
    list6l=[]
    for ch in listChar:
        if len(ch)<6:
            list6l.append(ch)
        elif len(ch)>=6:
            list6g.append(ch)
    print(listChar)
    print(list6l)
    print(list6g)

nbChar()
```

Exercice 10 : Déterminer si une année (introduit par l'utilisateur) est bissextile ou non. Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400.

Explication : Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile. Si elle est multiple de 4, on regarde si elle est multiple de 100. Si c'est le cas, on regarde si elle est multiple de 400. Si c'est le cas, l'année est bissextile. Si elle est multiple de 4, multiple de 100 et non multiple de 400 alors elle n'est pas bissextile. Si elle est multiple de 4 et non multiple de 100 alors, elle est bissextile.

Résolution

```
# on a utiliser une succession de clause if
def anneeType():
    try:
        annee=int(input("Entrez une année s'il vous plaît:"))
    except ValueError:
        print("Vous devez rentrer un nombre entier")
    if  (annee%4==0  &  annee%100==0  &  annee%400==0)|(annee%4==0  &
annee%100!=0):
        print("L'année", annee, " est bissextile")
    else :
        print("L'année", annee, " n'est pas bissextile")

anneeType()
```

Exercice 11 : Définissez une fonction changeCar(ch,ca1,ca2) qui remplace tous les caractères définis par ca1 par des caractères définis par ca2 dans la chaîne de caractères ch. Exemples de la fonctionnalité attendue :

phrase = 'Ceci est une toute petite phrase.'

print(changeCar(phrase, ' ', '--')) # renvoie Ceci--est--une*toute--petite--phrase.

Résolution

```
# On utilise tout simplement la fonction replace
def changeCar(ch,ca1,ca2):
    print(ch)
    ch=ch.replace(ca1, ca2) # attention, il faut créer ch afin de prendre en compte la modification lors de print
    print(ch)
# appel
phrase = 'Ceci est une toute petite phrase.'
val1=' '
val2='--'
changeCar(phrase, val1,val2)
```

Exercice 12 : 1-Ecrire une petite fonction trouve() qui fera exactement le contraire de ce que fait l'opérateur d'indexage (c'est-à-dire les crochets [ ]). Au lieu de partir d'un index donné pour retrouver le caractère correspondant, cette fonction devra retrouver l'index correspondant à un caractère donné. En d'autres termes, il s'agit d'écrire une fonction qui attend deux arguments : le nom de la chaîne à traiter et le caractère à trouver. La fonction doit fournir en retour l'index du premier caractère de ce type dans la chaîne. Ainsi par exemple, l'instruction : print(trouve("Juliette & Roméo", "&")) devra afficher : 9. Attention : il faut penser à tous les cas possibles. Il faut notamment veiller à ce que la fonction renvoie une valeur particulière (par exemple la valeur -1) si le caractère recherché n'existe pas dans la chaîne traitée. Les caractères accentués doivent être acceptés.

2- Améliorez la fonction de l'exercice précédent en lui ajoutant un troisième paramètre : l'index à partir duquel la recherche doit s'effectuer dans la chaîne. Ainsi par exemple, l'instruction : print(trouve ("César & Cléopâtre", "r", 5)) devra afficher : 15 (et non 4 !).

Résolution

```
# D'abord, on va utiliser la fonction find()
def trouve(text, ch):
    print(text.find(ch)) # définir l'index du caractère ch dans le texte text
# appel
phrase = 'Juliette & Roméo'
val='&'
trouve(phrase, val)
```

```
# On peut aussi ajouter les options start ou end pour indiquer la plage de recherche
def trouve(text, ch, start):
    print(text.find(ch,int(start))) # penser aussi à utiliser .index() qui renvoie l'index du premier élément.
# appel
phrase = 'César & Cléopâtre'
val='r'
debut=5
#trouve(phrase, val, debut) # on recherche le caractère r à partir de la 5 ième position et renvoyer l'index du premier r trouvé
```

Exercice 13: Dans un conte américain, huit petits canetons s'appellent respectivement : Jack, Kack, Lack, Mack, Nack, Oack, Pack et Qack. Écrivez un petit script qui génère tous ces noms à partir des deux chaînes suivantes : prefixes = 'JKLMNOP' et suffixe = 'ack'

Résolution

```
def nom():
    prefixes = 'JKLMNOP'
    suffixe = 'ack'
    listNom=[]
    for p in prefixes:
        listNom.append(p+suffixe)
    print(listNom)
# appel
nom()
```

Exercice 14 : Dans un script, écrivez une fonction qui recherche le nombre de mots contenus dans une phrase donnée.

Résolution

```
# Pour répondre à cette question, on peut utiliser la fonction split(). Ensuite compte le nombre d'élement en utilisant len().
phrase="je fais un test pour voir si ça marche"
#print(len(phrase.split()))
#NB: La fonction split() ne prend pas en compte l'espace par défaut. Mais on peut modifier ce comportement en indiquant un séparateur
```

Exercice 15 : Soit la liste suivante : ['Jean-Michel', 'Marc', 'Vanessa', 'Anne', 'Maximilien', 'Alexandre-Benoît', 'Louise']

Écrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant.

Résolution

```
# On peut appliquer la fonction map(len ) et former une liste des longeur. Ensuite afficher les caractère et leur longueur dans un dictionnaire
def charLenFunc():
    charlist=['Jean-Michel', 'Marc', 'Vanessa', 'Anne', 'Maximilien', 'Alexandre-Benoît', 'Louise']
    charlen=list(map(len, charlist))
    for i in range(0,6):
        print("(",charlist[i],",",charlen[i], ")")

charLenFunc()
```

Exercice 16 : Écrivez un script qui recherche le mot le plus long dans une phrase donnée (l'utilisateur du programme doit pouvoir entrer une phrase de son choix).

Résolution

```
def longWord():
    phrase=str(input("Entrez une phrase de votre choix:"))
    while phrase=="" or phrase==" ": # Controler si l'utilisateur rentre du vide.
        print("Vous devez rentrer une valeur valide")
        phrase=str(input("Entrez une phrase de votre choix:"))
    listWords=phrase.split() # décomposer la phrase en listes de mots
    listWordslen=list(map(len, listWords)) # la liste contenant les longueurs de chaque mot
    from functools import reduce
    maxlen=reduce(max, listWordslen) # récupérer la valeur maximale
    listLongWord=[]
    for w in listWords:
        if len(w)==maxlen:
            listLongWord.append(w) # rassembler dans une liste les mots dont la longueur est égale au max (il peut y en avoir plusieurs de même longueur)
    setLongWord=set(listLongWord)  # transformer cette liste en set pour supprimer d'éventuelles doublons
    words=",".join(list(setLongWord))
    print("Le(s) mot(s) le(s) plus long(s) est(sont):", words)
```

```
#appel
longWord()
```

Exercice 17 : Écrivez un script capable d'afficher la liste de tous les jours d'une année (calendrier imaginaire), laquelle commencerait un jeudi. Votre script utilisera lui-même trois listes : une liste des noms de jours de la semaine, une liste des noms des mois, et une liste des nombres de jours que comportent chacun des mois (ne pas tenir compte des années bissextiles).

Exemple de sortie :

jeudi 1 janvier vendredi 2 janvier samedi 3 janvier dimanche 4 janvier … et ainsi de suite, jusqu'au jeudi 31 décembre.

Résolution

```
def calendarIm():
# définition des paramètres
    listJour=[  'Jeudi',  'Vendredi',  'Samedi',  'Dimanche',  'Lundi',
'Mardi', 'Mercredi']
    listMois=[ 'Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin',
'Juillet', 'Aout', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
    listNbJoursMois = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    num=list(range(1,13)) # pour s'arreter à 12 avec range, il faut
indique 13 (i.e n+1)
    calendarMois=[]
    calendarJourNum=[]
    calendarAll=[]
    for n in num : # Extension des trois listes pour couvrir toute
l'année
        for i in list(range(1,listNbJoursMois[n-1]+1)):
            calendarJourNum.append(str(i))  #  Remplissage  des  jours
numériques
            calendarMois.append(listMois[n-1]) # remplissage des mois
    #print(calendarJourNum)
    #print(calendarMois)
    calendarJourCh=listJour*int((len(calendarJourNum)//7)+1)          #
démultiplication de la liste des jours pour pouvoir couvrir toute
l'année
    #print(calendarJourCh)
    #print(len(calendarJourCh))
    for  i  in  list(range(0,len(calendarJourNum))):  #  Formation  des
dates complètes (jour par kjour)
        globals()["date%s"%i]=[]
        globals()["date%s"%i].append(calendarJourCh[i])
        globals()["date%s"%i].append(calendarJourNum[i])
```

```
        globals()["date%s"%i].append(calendarMois[i])
        globals()["date%s"%i]="     ".join(globals()["date%s"%i])     #
formation de la date complète en texte
        #print(globals()["date%s"%i])
    for  i  in  list(range(0,len(calendarJourNum))):  #  Formation  du
calendrier complet (en associant les dates dans un liste)
        calendarAll.append(globals()["date%s"%i])
    calendarAll=",".join(calendarAll)   #   formation   du   calendrier
complet avec les dates séparées par des virgules
    print(calendarAll)
# Appel fonction
calendarIm()
```

Exercice 18 : Un nombre premier est un nombre qui n'est divisible que par un et par lui-même. Écrivez un programme qui établit la liste de tous les nombres premiers compris entre 1 et 1000, en utilisant la méthode du crible d'Eratosthène :

• Créez une liste de 1000 éléments rempli de 1.

• Parcourez cette liste à partir de l'élément d'indice 2 : si l'élément analysé possède la valeur 1, mettez à zéro tous les autres éléments de la liste, dont les indices sont des multiples entiers de l'indice auquel vous êtes arrivé. Lorsque vous aurez parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés.

En effet : A partir de l'indice 2, vous annulez tous les éléments d'indices pairs : 4, 6, 8, 10, etc. Avec l'indice 3, vous annulez les éléments d'indices 6, 9, 12, 15, etc., et ainsi de suite. Seuls resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.

Résolution

```
def npremier():
    table1=[]
    for i in list(range(1,1001)):
        table1.append(1) # Initialisation de la table
    print(table1)

    for   ind1,   val1   in   enumerate(table1[2:],2):   #   Commencer
l'énumération avec l'indice d'ordre 2.
        if val1==1:                                   # rechercher les
multiples de ind1
            for ind2, val2 in enumerate(table1[ind1+1:],ind1+1):
                if ind2%ind1==0:
                   table1[ind2] = 0
    print(table1)
```

```
    listnombre=[]                              # On va récupérer les
index des valeurs 1 pour former la liste des nombres premiers
(commençant par 2)
    for i, v in enumerate(table1[2:],2):
        if v==1:
            listnombre.append(i)
            #listnombre.append(str(i))
    #listnombre=",".join(listnombre)
    print(listnombre)
# appel
npremier()


# On pouvait aussi utiliser d'autres méthodes (comme les deux
approches ci-dessous)
#Méthode 2: (peu optimale mais assez intuitive)
## Pour chaque nombre de 2 à 100, calculez le reste de la division
entière (avec l'opérateur modulo %) depuis 1 jusqu'à lui-même. Si
c'est un nombre premier, il aura exactement deux nombres pour lesquels
le reste de la division entière est égal à 0 (1 et lui-même). Si ce
n'est pas un nombre premier, il aura plus de deux nombres pour
lesquels le reste de la division entière est égal à 0.

##   Méthode 3: (plus optimale et plus rapide, mais un peu plus
compliquée)
Vous pouvez parcourir tous les nombres de 2 à 100 et vérifier si ceux-
ci sont composés, c'est-à-dire qu'ils sont le produit de deux nombres
premiers. Pratiquement, cela consiste à vérifier que le reste de la
division entière (opérateur modulo %) entre le nombre considéré et
n'importe quel nombre premier est nul. Le cas échéant, ce nombre n'est
pas premier.
```

Exercice 19: Écrivez un script qui crée un mini-système de base de données fonctionnant à l'aide d'un dictionnaire, dans lequel vous mémoriserez les noms d'une série de copains, leur âge et leur taille. Votre script devra comporter deux fonctions : la première pour le remplissage du dictionnaire, et la seconde pour sa consultation. Dans la fonction de remplissage, utilisez une boucle pour accepter les données entrées par l'utilisateur. Dans le dictionnaire, le nom de l'élève servira de clé d'accès, et les valeurs seront constituées de tuples (âge, taille), dans lesquels l'âge sera exprimé en années (donnée de type entier), et la taille en mètres (donnée de

type réel). La fonction de consultation comportera elle aussi une boucle, dans laquelle l'utilisateur pourra fournir un nom quelconque pour obtenir en retour le couple « âge, taille » correspondant. Le résultat de la requête devra être une ligne de texte bien formatée, telle par exemple : « Nom : Jean Dhoute - âge : 15 ans - taille : 1.74 m ». Pour obtenir ce résultat, servez-vous du formatage des chaînes de caractères décrit à la page 138.

Résolution

```
# esssayons d'écrire un programme plus général
def datasys():
    # Registre: prénom, Nom, age, poids tailles
    registre=                                {'001A':['JEAN','MARTIN',
25,70,1.75],'001B':['PAUL','BERNARD',30,65,1.80],'001C':['PIERRE','DUB
OIS',35,75,1.65],'001D':['BAPTISTE','THOMAS', 28,80,1.90] }
    print("Que souhaitez-vous faire ?")
    objet=int(input("Si vous souhaitez consulter la base, tapez 1. Si
vous voulez ajouter des enregistrement, tapez 2:"))
    if objet==1: # Consultation de la base
        identMes=input("Disposez-vous  du  numéro  d'identification  ?
Tapez O pour oui et N pour non:")
        if identMes.upper()=="O":
            ident=str(input("Entrer l'identifiant:"))
            # Recherche par l'identifiant
            if '%s'%(ident.upper()) in registre:

globals()["values%s"%(ident.upper())]=registre['%s'%(ident.upper())]
                prenom=globals()["values%s"%(ident.upper())][0]
                nom=globals()["values%s"%(ident.upper())][1]
                age=globals()["values%s"%(ident.upper())][2]
                poids=globals()["values%s"%(ident.upper())][3]
                taille=globals()["values%s"%(ident.upper())][4]
                print("L'individu recherché est:")
                print("{0}  {1},{2}ans,{3}kilos,  {4}m".format(prenom,
nom, age, poids, taille))
            else:
                print("Il  n'y  aucun  individu  correspondant  à  cet
identifiant")
        elif identMes.upper()=="N":
            nameMes=input("Disposez-vous du nom de famille ? Tapez O
pour oui et N pour non:")
            if nameMes.upper()=="O":
                name=str(input("Entrer le nom:"))
                # partie à ajouter
            elif nameMes.upper()=="N":
                fnameMes=input("Disposez-vous de son prénom ? Tapez O
pour oui et N pour non:")
                if fnameMes.upper()=="O":
                    fname=str(input("Entrer le prénom:"))
                    # partie à ajouter
```

```
                if fnameMes.upper()=="N":
                    print("Vous devez fournir au moins un critère de
recherche")
                    abandMes=input("Souhaitez-vous abandonner ? Tapez
O pour oui et N pour recommencer la recherche:")
                    if abandMes.upper()=="O":
                      print(" ")
                    elif abandMes.upper()=="N":
                        #(remettre le début ici)
                        print(" mettre les instruction ici")
    elif objet==2: # Ajout d'enregistrements
        # partie à ajouter
        print("Instruction à ajouter ici")
# appel
datasys()
```

Exercice 20 : Suite de Fibonacci. La suite de Fibonacci est une suite mathématique qui porte le nom de Leonardo Fibonacci, un mathématicien italien du XIIIe siècle. Initialement, cette suite a été utilisée pour décrire la croissance d'une population de lapins mais elle est peut également être utilisée pour décrire certains motifs géométriques retrouvés dans la nature (coquillages, fleurs de tournesol...). Par définition, les deux premiers termes de la suite de Fibonacci sont 0 et 1. Ensuite, le terme au rang n est la somme des nombres aux rangs n - 1 et n - 2. Par exemple, les 10 premiers termes de la suite de Fibonacci sont 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Écrivez un script qui construit la liste des 20 premiers termes de la suite de Fibonacci puis l'affiche.

Résolution

```
def suiteFib():
    s=[0,1]
    for i,v in enumerate(list(range(2,20)),2):
        print(i)
        a=int(s[i-2])
        b=int(s[i-1])
        s.append(a+b)
        print(s)
    print(s)
suiteFib()
```

Exercice 21 : Conjecture de Syracuse.

La conjecture de Syracuse est une conjecture mathématique qui reste improuvée à ce jour et qui est définie de la manière suivante. Soit un entier positif n. Si n est pair, alors le diviser par 2. Si il est impair, alors le multiplier par 3 et lui ajouter 1. En répétant cette procédure, la suite de nombres atteint la valeur 1 puis se prolonge indéfiniment par une suite de trois valeurs triviales appelée cycle trivial. Jusqu'à présent, la conjecture de Syracuse, selon laquelle depuis n'importe quel entier positif la suite de Syracuse atteint 1, n'a pas été mise en défaut. Par exemple, les premiers éléments de la suite de Syracuse pour un entier de départ de10 sont : 10, 5, 16, 8, 4, 2, 1... Écrivez un script qui, partant d'un entier positif n, crée une liste des nombres de la suite de Syracuse. Avec différents points de départ (n), la conjecture de Syracuse est-elle toujoursvérifiée ? Quels sont les nombres qui constituent le cycle trivial ?

Remarque 1 : pour cet exercice, vous avez besoin de faire un nombre d'itérations inconnu pour que la suite de Syracuse atteigne 1 puis entame son cycle trivial. Vous pourrez tester votre algorithme avec un nombre arbitraire d'itérations, typiquement 20 ou 100, suivant votre nombre n de départ.

Remarque 2 : un nombre est pair lorsque le reste de sa division entière (opérateur modulo %) par 2 est nul.

Résolution

```
def syracuseConject():
    x=int(input("Entrer un nombre entier supérieur à 1:"))
    iterN=int(input("Entrer le nombre d'itérations souhaité (n>1):"))
    iter=1
    listN=[x]
    while iter<= iterN:
        if x%2==0:
            x=x//2
            listN.append(x)
        else :
            x=(x*3)+1
            listN.append(x)
            if x%2==0:
                x=x//2
                listN.append(x)
        iter=iter+1
    print(listN)
syracuseConject()
```

Exercice 22: Mot composable. Un mot est composable à partir d'une séquence de lettres si la séquence contient toutes les lettres du mot. Comme au Scrabble, chaque lettre de la séquence ne peut être utilisée qu'une seule fois. Par exemple, coucou est composable à partir de uocuoceokzefhu.

Écrivez un script qui permet de savoir si un mot est composable à partir d'une séquence de lettres. Testez le avec différents mots et séquences.

Remarque : dans votre script, le mot et la séquence de lettres seront des chaînes de caractères.

Résolution

```
def composable():
##    mot=str(input("Entrer le mot à analyser:"))
##    sequence=str(input("Entrer la séquence à scanner:"))
    mot="coucou"
    sequence="uocuoceokzefhu"
    message="Le mot "+ str(mot)+ " est composable à partir de la
séquence "+ str(sequence)

    if len(mot)>len(sequence):
        print("Le mot "+ str(mot)+ "n'est pas composable à partir de
la séquence "+ str(sequence))
    else:
        for i,v in enumerate(list(mot),1):
            if v in sequence:
                 if mot.count(v)>sequence.count(v):
                    print("Le mot "+ str(mot)+ "n'est pas composable à
partir de la séquence "+ str(sequence))
                    break
                 else:
                    #message=message
                    if i==len(mot):
                        print(message)
            else:
                print("Le mot "+ str(mot)+ "n'est pas composable à
partir de la séquence "+ str(sequence))
# Appel
composable()
```

Exercice 23: Alphabet et pangramme. Les codes ASCII des lettres minuscules de l'alphabet vont de 97 (lettre a) à 122 (lettre z). La fonction chr() prend en argument un code ASCII sous forme d'une entier et renvoie le caractère correspondant. Ainsi chr(97) renvoie 'a', chr(98) renvoie 'b' et ainsi de suite. Écrivez un script qui construit une chaîne de caractères contenant toutes les lettres de l'alphabet.

Un pangramme est une phrase comportant au moins une fois chaque lettre de l'alphabet. Par exemple, « Portez ce vieux whisky au juge blond qui fume » est un pangramme. Modifiez le script précédent pour déterminer si une chaîne de caractères est un pangramme ou non. Pensez à vous débarasser des majuscules le cas échéant. Testez si les expressions suivantes sont des pangrammes : « Monsieur Jack vous dactylographiez bien mieux que votre amiWolf », « Buvez de ce whisky que le patron juge fameux ».

Résolution

```
def alphab():
    listAlphab=[]
    for i in list(range(97,123)):
        listAlphab.append(chr(i).upper())
    listAlphab="".join(listAlphab)
    print(listAlphab)
#Appel
alphab()


# Pour montrer si un texte est un pangramme, on va simplement vérifié si chaque lettre de l'alphabet se trouve dans le texte en question. Ainsi on a:
def pangramme():
    #phrase=str(input("Entrer une phrase à analyser:"))
    #phrase="Monsieur Jack vous dactylographiez bien mieux que votre ami Wolf"
    #phrase="Buvez de ce whisky que le patron juge fameux"
    #phrase ="Portez ce vieux whisky au juge blond qui fume"

    sequence="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    message="La phrase indiquée est un pangramme"

    for i,v in enumerate(list(sequence.upper()),1):
        if not v in phrase.upper():
            print("La phrase indiquée n'est pas un pangramme")
            break
        else:
            #message=message
            if i==len(sequence):
                print(message)
# Appel
pangramme()
```

Exercice 24: Jeu de La Roulette:

Règles du jeux: Le joueur mise sur un numéro compris entre 0 et 49 (50 numéros en tout). En choisissant son numéro, il y dépose la somme qu'il souhaite miser. La roulette est constituée

de 50 cases allant naturellement de 0 à 49. Les numéros pairs sont de couleur noire, les numéros impairs sont de couleur rouge. Le croupier lance la roulette, lâche la bille et quand la roulette s'arrête, relève le numéro de la case dans laquelle la bille s'est arrêtée. Le numéro sur lequel s'est arrêtée la bille est, naturellement, le numéro gagnant. Si le numéro gagnant est celui sur lequel le joueur a misé (probabilité de 1/50, plutôt faible), le croupier lui remet 3 fois la somme misée. Sinon, le croupier regarde si le numéro misé par le joueur est de la même couleur quele numéro gagnant (s'ils sont tous les deux pairs ou tous les deux impairs). Si c'est le cas, le croupier lui remet 50 % de la somme misée. Si ce n'est pas le cas, le joueur perd sa mise. Dans les deux scénarios gagnants vus ci-dessus (le numéro misé et le numéro gagnant sont identiques ou ont la même couleur), le croupier remet au joueur la somme initialement misée avant d'y ajouter ses gains. Cela veut dire que, dans ces deux scénarios, le joueur récupère de l'argent. Il n'y a que dans le troisième cas qu'il perd la somme

Résolution:

```
##  misée 1.
# Démarche:
# 1- le jour va d'abord entrer deux valeurs: le choix du numéro et le montant misé
# 2- la mise va faire un tirage aléatoire du numéro gagant
# 3- Ensuite on analyse les résultats pour déterminier le gain(perte) potentiel.

def jeu():
    # Pour simplifier, on va ignorer tous les controles de validité
    numMise=int(input("Choisir un numéro entre 0 et 49:"))
    if numMise%2==0:
        numMiseCoul="Noire"
    else:
        numMiseCoul="Rouge"
    valMise=float(input("Indiquer le montant à miser:"))

    import random
    numTire=random.randint(0,49)
    if numTire%2==0:
        numTireCoul="Noire"
    else:
        numTireCoul="Rouge"

    if numMise==numTire:
        valGain=valMise*3
    elif (numMise!=numTire)&(numMiseCoul==numTireCoul):
        valGain=valMise*0.5
    else:
        valGain=0
    gainNet=valGain-valMise
    print("Le numéro que vous avez choisi est: %i"%numMise)
```

```
    print("Le numéro gagnant est: %i"%numTire)
    print("Le  montant  que  vous  avez  misé  est:  %.2f"%valMise  +  "
euro(s)")
    print("Votre gain brut est: %.2f"%valGain + " euro(s)")
    print("Votre gain net est: %.2f"%gainNet + " euro(s)")
#Appel
jeu()
```

Exercice 25 : Écrivez un script qui permette de créer et de relire aisément un fichier texte. Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite il lui proposera le choix, soit d'enregistrer de nouvelles lignes de texte, soit d'afficher le contenu du fichier.

L'utilisateur devra pouvoir entrer ses lignes de texte successives en utilisant simplement la touche <Enter> pour les séparer les unes des autres. Pour terminer les entrées, il lui suffira d'entrer une ligne vide (c'est-à-dire utiliser la touche <Enter> seule). L'affichage du contenu devra montrer les lignes du fichier séparées les unes des autres de la manière la plus naturelle (les codes de fin de ligne ne doivent pas apparaître).

Résolution

```
def textproc():
    import os, sys,shutil
    os.chdir("D:/fichier    sur    disque/vista/mes    fichiers/PYTHON
TUTORIAL/applications/data" )
    print("Que souhaitez-vous faire ?")
    objet=int(input("Taper  1  si  vous  souhaitez  ajouter  des  textes.
Taper 2 si vous voulez consulter le contenu du fichier:"))
    while objet!=1 and objet!=2:
        print("Vous devez rentrer 1 ou 2 pour choisir")
        objet=int(input("Taper 1 si vous souhaitez ajouter des textes.
Taper 2 si vous voulez le contenu du fichier:"))

    fichier=str(input("Taper le nom du fichier avec son extention. Ex:
myfile.text:"))
    print("Attention,  si  le  fichier  indiqué  n'existe  pas,  il  sera
crée")
    if objet==1: # ajout texte
        text=str(input("Entrer  le  texte  que  vous  voulez  ajouter  au
fichier"))

        with open('%s'%fichier, 'a') as x:  # Ouverture en mode append
            x.write(+"\n"+"%s"%text)
    elif objet==2: # consultation du fichier
        with open('%s'%fichier, 'r') as x: # Ouverture en mode read
```

```
            for i in x:
                print(i.strip())
#appel
textproc() # tester avec le fichier fromageUn.txt pour consultation et
avec mytext_ajout.txt pour ajout
```

Exercice 26 : Écrivez un script qui génère automatiquement un fichier texte contenant les tables de multiplication de 2 à 30 (chacune d'entre elles incluant 20 termes seulement).

Résolution

```
# Résolution
# défintion de la fonction multipli7
def multipli():
    import os, sys,shutil
    os.chdir("D:/fichier sur disque/vista/mes fichiers/PYTHON
TUTORIAL/applications/data" )
    i=2
    with open('mutitable.txt', 'w') as x:  # Ouverture en mode write
qui ecrase le fichier existant
        while i<=30:
            j=1
            x.write("\n"+ "Table de mutiplication de %i"%i)
            while j<=20:
                x.write("\n"+"%i"%i+ " x "+"%i"%j + " =
"+"{0}".format(i*j))
                j=j+1
            i=i+1

#appel de la fonction
multipli()
 #vérification
with open('mutitable.txt', 'r') as x: # Ouverture en mode read
            for i in x:
                print(i.strip())
```

Exercice 27 : Écrivez un script qui compare les contenus de deux fichiers et signale la première différence rencontrée.

Résolution

Idée: Deux fichiers sont identiques lorsqu'ils ont les mêmes éléments situés aux mêmes positions. Le premier élement du fichier 1 correspond au premier élement fichier 2. le second au seconde, etc..

On va donc essayer de comparer élement par élement (c'est à dire ligne par ligne).Pour cela, on va d'abord stocker le contenu de chaque fichier dans des objets list

Ainsi, si les deux listes, n'ont pas la même dimension (len()), on conclura à une différence. Et si elles ont les mêmes longueurs, on va comparer éléments par élément

Si tous les éléments sont égaux, on afficheara un messge "que les deux fichiers sont pareils" et et s'il ya des différence au niveau deux élements, on affichera un message

que les deux fichiers ne sont pas paeils et on affichera les deux élemenst différents.

```
def comparefile():
    myfile1=input("Entrer le nom du premier fichier avec son extension:")
    myfile2=input("Entrer le nom du deuxième fichier avec son extension:")
    import os, sys,shutil
    os.chdir("D:/fichier sur disque/vista/mes fichiers/PYTHON TUTORIAL/applications/data" )
    import re
    # Récupération du texte du premier fichier
    mylinelist1 = [line.strip() for line in open('%s'%myfile1, 'r')] # on pouvait aussi utiliser la méthode classique de splitlines()
    mylinelist1 = [re.split(r'\t+', line) for line in mylinelist1] # pour former les listes des listes sans tabulation \t
    # On va maintenant réconstituer les lignes en remplaçant les tabulation par des espaces.
    index=len(mylinelist1) # Nombre d'éléments(lignes) dans mylinelist
    for k in list(range(0,index)):
        globals()["l%s"%k]=mylinelist1[k]  # Récuperer les sous-éléments correspondant à k et faire join() avec espace comme séparateur
        globals()["l%s"%k]=" ".join(globals()["l%s"%k])
        mylinelist1[k]=globals()["l%s"%k] # les tabulations remplacées par des espaces
    print(mylinelist1)

    # Récupération du texte du deuxième fichier
    mylinelist2 = [line.strip() for line in open('%s'%myfile2, 'r')] # on pouvait aussi utiliser la méthode classique de splitlines()
    mylinelist2 = [re.split(r'\t+', line) for line in mylinelist2] # pour former les listes des listes sans tabulation \t
```

```
    # On va maintenant réconstituer les lignes en remplaçant les
tabulation par des espaces.
    index=len(mylinelist2) # Nombre d'éléments(lignes) dans mylinelist
    for k in list(range(0,index)):
        globals()["l%s"%k]=mylinelist2[k]   #   Récuperer   les   sous-
éléments  correspondant  à  k  et  faire  join()  avec  espace  comme
séparateur
        globals()["l%s"%k]=" ".join(globals()["l%s"%k])
        mylinelist2[k]=globals()["l%s"%k] # les tabulations remplacées
par des espaces
    print(mylinelist2)

    dim1=len(mylinelist1)
    dim2=len(mylinelist2)
    # Comparaison
    if dim1!=dim2:
        print("Les deux fichiers ne sont pas identiques car ils n'ont
pas le même nombre de mots")
    else:
        n=0
        while n<dim1:
            if mylinelist1[n]!=mylinelist2[n]:
                val1=mylinelist1[n]
                val2=mylinelist2[n]
                print("Les  deux  fichiers  ne  sont  pas  identiques.  La
première différence se trouve sur la {0} ième ligne du premier fichier
<<{1} >>  \
qui   est   différent   de   la   {0}   ième   ligne   du   second   fichier
<<{2}>>".format(n,val1, val2)   )
                break
            else:
                if n==dim1-1:
                    print("Les deux fichiers sont identiques")
            n=n+1
comparefile() # Pour tester la fonction on va utiliser les trois
fichiers:fromageUTF8-1.txt,  fromageUTF8-2.txt  et  fromageUTF8-3.txt.
Les deux derniers à comparer au premeir
```

Exercice 28 : À partir de deux fichiers préexistants A et B, construisez un fichier C qui contienne alternativement un élément de A (ligne), un élément de B (ligne), un élément de A… et ainsi de suite jusqu'à atteindre la fin de l'un des deux fichiers originaux. Complétez ensuite C avec les éléments restant sur l'autre.

Résolution

```
def copyaltern():
    myfile1=input("Entrer  le  nom  du  premier  fichier  avec  son
extension:")
```

```
    myfile2=input("Entrer  le  nom  du  deuxième  fichier  avec  son
extension:")
    import os, sys,shutil
    os.chdir("D:/fichier    sur    disque/vista/mes    fichiers/PYTHON
TUTORIAL/applications/data" )
    x1 = open('%s'%myfile1, 'r')
    x2 = open('%s'%myfile2, 'r')
    ch1=x1.readline()
    ch2=x2.readline()
    with open('fileC.txt', 'w') as y:  # Ouverture en mode write qui
ecrase le fichier existant
        while ch1!="" or ch2!="":
            if ch1!="" and ch2!="":
                y.write("%s"%ch1+"\n")
                y.write("%s"%ch2+"\n")
                ch1=x1.readline()
                ch2=x2.readline()
            elif ch1=="" and ch2!="":
                y.write("%s"%ch2+"\n")
                ch2=x2.readline()
            elif ch1!="" and ch2=="":
                y.write("%s"%ch1+"\n")
                ch1=x1.readline()
    x1.close();x2.close()

copyaltern() # on va le tester sur deux fichiers: fichierStar.txt et
fichierTiret.txt
#voir les résultat dans le fichier fileC.txt
#Pour améliorer l'affichage en print, il faut modifier l'encodage(voir
le cours sur l'encodag)
```

Exercice 29 : Écrivez un script qui permette de créer un fichier texte dont les lignes contiendront chacune les noms, prénom, adresse, code postal et no de téléphone de différentes personnes (considérez par exemple qu'il s'agit d'une fiche des clients).

Résolution

```
def ficheclients():
    import os, sys,shutil
    os.chdir("D:/fichier    sur    disque/vista/mes    fichiers/PYTHON
TUTORIAL/applications/data" )

    nom=input("Entrer le nom:")
    prenom=input("Entrer le prénom:")
    adresse=input("Entrer l'adresse:")
    codepost=input("Entrer le code postal:")
    ville=input("Entrer la ville:")
    tel=input("Entrer le numéro de téléphone:")
```

```
    email=input("Entrer l'émail:")
    message=int(input("Souhaitez-vous ajouter un autre enregistement ? 
Si oui tapez 1, sinon tapez 0:"))
    with open('ficheclients.txt','a') as x: # mode append pour ne pas 
écraser le fichier lors des prochaines ouverture
        y=open('ficheclients.txt','r')  #  Pour  vérifier  s'il  existe 
déja des lignes (afin de prévoir une nouvelle ligne \n pour le nom)
        ch=y.readline()
        if ch=="":
            x.write("%s"%nom+  ";  ")  #  Séparateur  "point-virgule  et 
espace combiné" ; car l'espace seule n'est pas bien adapté ici.
        else:
            x.write("\n"+"%s"%nom+  ";  ")  #  Pour  mettre  sur  une 
nouvelle ligne au cas il y aurait un nouveau élements
        y.close()
        x.write("%s"%prenom+ "; ")
        x.write("%s"%adresse+ "; ")
        x.write("%s"%codepost+ "; ")
        x.write("%s"%ville+ "; ")
        x.write("%s"%email+ "; ")
        #  NB:  On  a  utilisé  Séparateur  "point-virgule  et  espace 
combiné" ; car l'espace seule n'est pas bien adapté ici. # nouvelle 
ligne
    while message==1:
        nom=input("Entrer le nom:")
        prenom=input("Entrer le prénom:")
        adresse=input("Entrer l'adresse:")
        codepost=input("Entrer le code postal:")
        ville=input("Entrer la ville:")
        tel=input("Entrer le numéro de téléphone:")
        email=input("Entrer l'émail:")
        message=int(input("Souhaitez-vous     ajouter     un     autre 
enregistement ? Si oui tapez 1, sinon tapez 0:"))
        with open('ficheclients.txt','a') as x: # mode append pour ne 
pas écraser le fichier lors des prochaines ouverture
                x.write("\n"+"%s"%nom+ "; ")
                x.write("%s"%prenom+ "; ")
                x.write("%s"%adresse+ "; ")
                x.write("%s"%codepost+ "; ")
                x.write("%s"%ville+ "; ")
                x.write("%s"%email+ "; ")
ficheclients()
# test sur le fichier crée
x = open('ficheclients.txt', 'r')
for i in x :
    print(i.strip())
x.close()
# Afficher dans un dataframe
import  pandas
```

```
pandas.set_option('expand_frame_repr', False) # augmente le nombre de variable par page
x=pandas.read_csv("ficheclients.txt",sep=';', header=None, encoding ='Latin-1') # Si la table contient des caractères accentués on utilise encoding ='Latin-1' (penser aussi à d'autre type d'encodage)
print(x)
```

Exercice 30: Considérons que vous avez fait les exercices précédents et que vous disposez à présent d'un fichier contenant les coordonnées d'un certain nombre de personnes. Écrivez un script qui permette d'extraire de ce fichier les lignes qui correspondent à un code postal bien déterminé.

Résolution

```
# On va appliquer cet exercices sur le fichier ficheclients2.csv et afficher les clients qui se trouve au 75011
x = open('ficheclients2.csv', 'r')
for line in x :
    if "75011" in line: # Toutes les lignes contenant "75011"
        #print(line.strip())
        print(" ")
x.close()
# On peur aussi rassembler ces lignes et les afficher en dataframe sans séparateur ";".
```

Exercice 31 : Utiliser la page suivante: https://fr.wikipedia.org/wiki/Liste_des_capitales_du_monde pour extraire les capitales du monde.

Résolution

```
def capitalData():
    # méthode: on va d'abord ouvrir l'url
    import urllib.request
    with
urllib.request.urlopen("https://fr.wikipedia.org/wiki/Liste_des_capitales_du_monde") as u:  # ouverture de la page
        myHtmlText=u.read() # extraction du texte (non parsé)
    # On va décoder le texte. Pour cela, il faut connaitre le codec. Ce qui n'est pas toujours facile.
    # Mais en regardant le code sources de la page, on voit en en-tête que charset)"UTF-8". on va donc décoder en utilisant utf-8
    myHtmlText = myHtmlText.decode('utf-8')
```

```
    # On va parsing le fichier
    from bs4 import BeautifulSoup
    myParsedText = BeautifulSoup(myHtmlText , 'html.parser') # parsing
avec html.parser
    for i in myParsedText(["script", "style","form", "noscript"]): #
Supprimer les scripts et le styles
        i.extract()
    #print(myParsedText)
    cleanText = myParsedText.get_text() # Récupérer le texte brut (
avec encore les espaces)
    cleanText=cleanText.splitlines() # Découper en lignes sous formes
de listes
    lines = (line.strip() for line in cleanText) # Enlever les
premières espaces
    phraseList = (phrase.strip() for line in lines for phrase in
line.split("  ")) # Découper chaque ligne en mots (phrase) sous forme
de liste
    cleanText = '\n'.join(phrase for phrase in phraseList  if phrase)
# Supprimer les lignes vides et joindre des phrases avec \n comme
séparateur
    #print(cleanText)

    ### Première approche 1: Sélectionner toutes les lignes dans un
intervalle de texte bien défini (après avoir obervé le contenu du
texte)

    with open("myTextfile.txt","w",encoding='utf-8') as f:
        f.write((cleanText))
    with open('myTextfile.txt', 'r',encoding="utf-8") as x:
        textList=x.readlines()
    textList.remove('Union européenne (de facto)\n') # Cette valeur a
été attribuée à bruxelle en plus de la belgique
    istart=textList.index("Capitale\n") # index de la phrase de début
de sélection du texte à extraire (attention à la présence de \n qui
vient de la définition de cleanText)
    iend=textList.index("Capitales d'États dont la souveraineté est
contestée[modifier | modifier le code]\n") # index de fin de sélection
    capitale=[] # liste qui va récupérer toutes les noms de capitales
    etat=[] # liste qui va récupérer tous les noms des états
    n=istart
    while n<iend :
        if n%2!=0:
            capitale.append(textList[n].strip())
        else:
            etat.append(textList[n].strip())
        n=n+1
    #print(capitale)
    # On va créer un fichier csv pour regrouper les deux variables
    with open("capitaleData.csv","w",encoding='utf-8') as fic:
```

```
        for i in list(range(len(capitale))):
            vCapitale=capitale[i]
            vEtat=etat[i]
            fic.write("%s"%vCapitale+ ";")
            fic.write("%s"%(vEtat)+"\n") # remarquer la nouevlle ligne
ici
            print("%s"%vCapitale+ "(" + "%s"%(vEtat) +")"+"\n" )
    # Le fichier est maintenant enregsitré dans le répertoire sous
format csv (pouvant maintenant être appelé dans un dataframe avec
pandas.read_csv())
# appel de la fonction
capitalData()

### deuxième approche : Sélectionner la table correspondant aux
données et extraire les données ( approche à élaborer !)
```

Bibliographie

Biernat Éric et Lutz Michel, (2015), Data Science : fondamentaux et études de cas: Machine Learning avec Python et R, Edition Eyrolles

Bird Steven, Klein Ewan, et Loper Edward Natural(2009), Language Processing with Python, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

Cordeau Bob et Pointal Laurent, (2010),Une introduction à Python 3, disponible (04/10/2016) à http://hebergement.u-psud.fr/iut-orsay/Pedagogie/MPHY/Python/courspython3.pdf

Fuchs Patrick et Poulain Pierre, (2014), Cours de Python, Université Paris, Diderot, Disponible à http://www.dsimb.inserm.fr/~fuchs/python/cours_python.pdf (04/10/2016)

Le GOFF Vincent,(2011), Apprenez à programmer avec Python, Le site du Zero, ISBN : 979-10-90085-03-9.

Lutz Mark et Bailly Yves (2005), Python précis et concis, O'Reilly, 2e édition.

Martelli Alex, (2004), Python en concentré, Edition O'Reilly.

Rakotomalala Ricco, (2015), Machine learning avec scikit-learn, Programmation python, Université Lyon Lumière 2, disponible à http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_python.html

Summerfield Mark, (2009), Programming in Python 3, Addison-Wesley, 2e edition.

Swinnen, Gérard,(2010), Apprendre à programmer avec Python 3, Edition Eyrolles

Ziadé Tarek, (2007), Python : Petit guide à l'usage du développeur agile, Edition Dunod.

Ziadé Tarek,(2008), Expert Python Programming, Packt Publishing.

Ziadé Tarek,(2009), Programmation Python. Conception et optimisation, Eyrolles, 2e édition.