TPs

October 3, 2023

Contents

1 TP 1: Variables, Fonctions et Structure de control

1.1 Variables et Operations

En programmation, une variable est un espace de stockage nommé qui permet de stocker des données. En Python, un langage de programmation polyvalent et largement utilisé, il existe quatre grands types de variables qui nous permettent de stocker différents types de données. Comprendre ces types de variables est essentiel pour manipuler efficacement les données dans un programme Python. Les Quatre Types Principaux de Variables en Python

- int (nombre entier) : Les variables de type "int" sont utilisées pour stocker des nombres entiers. Par exemple, age = 25 stocke un nombre entier, dans ce cas l'âge de quelqu'un.
- float (nombre décimal) : Les variables de type "float" sont utilisées pour stocker des nombres décimaux. Par exemple, prix = 19.99 stocke un nombre décimal, ici un prix en euros.
- string (chaîne de caractères) : Les variables de type "string" sont utilisées pour stocker des séquences de caractères. Par exemple, nom = "Alice" stocke une chaîne de caractères, ici le nom d'une personne.
- bool (booléen) : Les variables de type "bool" sont utilisées pour stocker des valeurs booléennes, c'est-à-dire True (Vrai) ou False (Faux). Par exemple, est_majeur = True stocke une valeur booléenne, indiquant si quelqu'un est majeur ou non.

En Python, la déclaration d'une variable est simple. Vous pouvez attribuer une valeur à une variable en utilisant le signe égal (=). Par exemple, pour attribuer la valeur 42 à une variable nombre, vous feriez nombre = 42.

Il est important de noter que les noms de variables en Python doivent suivre certaines conventions, telles que ne pas commencer par un chiffre et ne contenir que des lettres, des chiffres et des underscores.

L'utilisation des variables est fondamentale en programmation, car elles nous permettent de stocker et de manipuler des données de manière dynamique. Dans ce cours, nous explorerons en détail chaque type de variable et comment les utiliser efficacement pour résoudre des problèmes de programmation.

Maintenant que nous avons une compréhension de base des variables en Python, explorons chacun de ces types en détail dans les sections suivantes.

[4]:
```
x = 3 # type int
y = 2.5 # type float
prenom = 'Pierre' # type string
z = True # type Bool

print(type(x),type(y), type(prenom),type(z))
```

```
<class 'int'> <class 'float'> <class 'str'> <class 'bool'>
```

1.1.1 Opérations Arithmétiques en Python

Les opérations arithmétiques sont un aspect fondamental de la programmation en Python. Elles permettent de réaliser des calculs mathématiques de base, ce qui est essentiel dans le développement

de tout type d'application. Dans cette section, nous explorerons les opérations arithmétiques de base en Python, notamment l'addition, la soustraction, la multiplication, la division et la puissance.

Python offre une syntaxe simple et intuitive pour effectuer ces opérations, ce qui en fait un langage de programmation populaire pour les calculs mathématiques et scientifiques, ainsi que pour des applications plus générales.

Nous aborderons les sujets suivants dans cette section : - Addition (+) - Soustraction (-) - Multiplication (*) - Division (/) - Opérations avec des nombres entiers - Utilisation de parenthèses pour contrôler l'ordre des opérations - Utilisation de la fonction `pow()` pour calculer les puissances

Comprendre comment utiliser ces opérations est essentiel pour manipuler des données numériques et résoudre des problèmes mathématiques à l'aide de Python.

[3]:
```
# Opérations arithmétiques
print('x + y =', x + y)
print('x - y =', x - y)
print('x / y =', x / y)
print('x // y =', x // y) # division entiere (tres utile pour les tableaux Numpy)
print('x * y =', x * y)
print('x ^ y =', x ** y) # x puissance y
```

```
x + y = 5.5
x - y = 0.5
x / y = 1.2
x // y = 1.0
x * y = 7.5
x ^ y = 15.588457268119896
```

1.1.2 Opérations de Comparaison en Python

Les opérations de comparaison sont essentielles en programmation pour évaluer des conditions et prendre des décisions en fonction de ces évaluations. En Python, vous pouvez comparer des valeurs et des expressions pour déterminer si elles sont égales, plus grandes, plus petites ou d'autres relations. Dans cette section, nous explorerons les opérations de comparaison en Python.

Voici les principales opérations de comparaison que nous allons aborder : - Égalité (==) : Vérifie si deux valeurs sont égales. - Inégalité (!=) : Vérifie si deux valeurs ne sont pas égales. - Supériorité (>) : Vérifie si une valeur est strictement supérieure à une autre. - Infériorité (<) : Vérifie si une valeur est strictement inférieure à une autre. - Supériorité ou égalité (>=) : Vérifie si une valeur est supérieure ou égale à une autre. - Infériorité ou égalité (<=) : Vérifie si une valeur est inférieure ou égale à une autre.

Ces opérations de comparaison sont couramment utilisées pour créer des expressions conditionnelles, des boucles et d'autres structures de contrôle de flux dans vos programmes Python.

[4]:
```
# Opérations de comparaison
print('égalité :', x == y)
print('inégalité :', x != y)
print('inférieur ou égal :', x <= y)
print('supérieur ou égal :', x >= y)
```

```
égalité : False
inégalité : True
inférieur ou égal : False
supérieur ou égal : True
```

1.1.3 Opérations Logiques en Python

Les opérations logiques sont un élément clé de la programmation en Python, utilisées pour évaluer des conditions complexes et prendre des décisions basées sur ces conditions. Les opérations logiques permettent de combiner et de manipuler des valeurs booléennes (Vrai ou Faux) de manière à construire des expressions conditionnelles puissantes. Dans cette section, nous explorerons les opérations logiques en Python.

Les opérations logiques principales que nous allons aborder sont les suivantes : - ET logique (`and`) : Vérifie si deux conditions sont simultanément vraies. - OU logique (`or`) : Vérifie si au moins l'une des deux conditions est vraie. - NON logique (`not`) : Inverse une condition booléenne.

Ces opérations logiques sont utilisées pour créer des expressions conditionnelles complexes qui permettent de contrôler le flux de votre programme. Elles sont essentielles pour prendre des décisions, exécuter certaines parties du code en fonction de conditions, et créer des boucles conditionnelles.

[5]:
```python
# Opérations Logiques
print('ET :', False and True)
print('OU :', False or True)
print('OU exclusif :', False ^ True)
```

```
ET : False
OU : True
OU exclusif : True
```

Note : Les opérations de comparaison et de logique utilisées ensemble permettent de construire des structures algorithmiques de bases (if/esle, while, . . .)

[6]:
```python
# Table Logiques de AND
print('  A     B   ||  A and B   ')
print('-------------------------')
print('False True  ||  ', False and True)
print('False False ||  ', False and False)
print('True  True  ||  ', True and True)
print('True  False ||  ', False and False)
```

```
  A     B   ||  A and B
-------------------------
False True  ||   False
False False ||   False
True  True  ||   True
True  False ||   False
```

1.2 Fonctions

Les fonctions sont des éléments fondamentaux de la programmation en Python. Elles permettent d'organiser et de structurer le code en le divisant en blocs de code réutilisables. Une fonction est un ensemble d'instructions qui effectuent une tâche spécifique et peuvent prendre des arguments en entrée pour personnaliser leur comportement. Les fonctions sont essentielles pour rendre le code plus lisible, modulaire et facile à entretenir.

Dans Python, une fonction est définie à l'aide du mot-clé def, suivi du nom de la fonction et de ses paramètres entre parenthèses. Elle peut avoir un corps de fonction qui contient les instructions à exécuter lorsque la fonction est appelée. Les fonctions peuvent également retourner des valeurs à l'aide du mot-clé return.

L'utilisation de fonctions permet de réduire la redondance du code en regroupant des actions similaires sous une seule fonction. Elles offrent également la possibilité de diviser un programme en morceaux plus petits et plus gérables, ce qui simplifie le débogage et la maintenance du code.

1.2.1 Les Fonctions Lambda en Python

Les fonctions lambda, également connues sous le nom de fonctions anonymes, sont un concept intéressant en Python. Contrairement aux fonctions définies avec def, les fonctions lambda sont des fonctions courtes et temporaires qui peuvent être créées sans avoir à nommer explicitement la fonction. Elles sont définies à l'aide du mot-clé lambda, suivi des paramètres entre parenthèses et d'une expression unique qui est évaluée et renvoyée comme résultat.

Présentation des Fonctions Lambda :

Les fonctions lambda sont souvent utilisées pour des opérations simples et ponctuelles, telles que les transformations de données dans une liste, les filtres ou les opérations mathématiques simples. Elles sont particulièrement utiles lorsque vous avez besoin d'une petite fonction pour effectuer une tâche spécifique, sans la complexité d'une fonction définie avec def.

Définition des Fonctions Lambda :

Une fonction lambda est définie comme suit :

```
lambda arguments: expression
```

- arguments : Les paramètres que la fonction lambda prend en entrée, séparés par des virgules.
- expression : L'expression unique qui est évaluée et renvoyée comme résultat lorsque la fonction lambda est appelée.

Voici un exemple simple d'une fonction lambda qui additionne deux nombres :

[7]:
```
# Exemple d'une fonction f(x) = x^2
f = lambda x : x**2

print(f(3))
```

```
9
```

[8]:
```
# Exemple d'une fonction g(x, y) = x^2 - y^2
g = lambda x, y : x**2 - y**2
```

```
print(g(4, 2))
```

```
12
```

[9]:
```
print(g(f(2), 2))
```

```
12
```

1.2.2 Fonction en Python

Une fonction en Python est une portion de code qui effectue une tâche spécifique ou une série de tâches. Les fonctions sont utilisées pour organiser et réutiliser du code en le divisant en blocs autonomes. Une fonction peut être conçue pour prendre des arguments en entrée, effectuer des opérations sur ces arguments, et renvoyer un résultat en sortie. Les fonctions sont définies à l'aide du mot-clé def, suivi du nom de la fonction et de ses paramètres entre parenthèses, le cas échéant.

Définition d'une Fonction :

Voici la structure de base de la définition d'une fonction en Python :

```
def nom_de_la_fonction(parametre1, parametre2, ...):
    # Instructions de la fonction
    # Ces instructions décrivent ce que la fonction fait
    # Elles peuvent inclure des calculs, des opérations, des boucles, etc.
    return resultat
```

- nom_de_la_fonction : Le nom que vous choisissez pour votre fonction.
- parametre1, parametre2, ... : Les paramètres (également appelés arguments) que la fonction peut accepter en entrée. Vous pouvez en spécifier un ou plusieurs, ou même aucun.
- return resultat : Facultatif. Si vous souhaitez que votre fonction renvoie un résultat, utilisez l'instruction return pour le spécifier. Le résultat peut être une valeur, un objet, une liste, etc.

[10]:
```
# une fonction a un nom, prend des entrées (arguments) et les transforme pour␣
↪retourner un résultat

def nom_de_la_fonction(argument_1, argument_2):
   restultat = argument_1 + argument_2
   return restultat

nom_de_la_fonction(3, 2)
```

[10]:
```
5
```

[11]:
```
# Exemple concret : fonction qui calcul l'energie potentielle d'un corps

def e_potentielle(masse, hauteur, g=9.81):
  energie = masse * hauteur * g
  return energie
```

```
# ici g a une valeur par défaut donc nous ne sommes pas obligé de lui donner une␣
 ↪valeur
e_potentielle(masse=10, hauteur=10)
```

[11]: 981.0

a notion de docstring, ou chaîne de documentation, est un concept important en programmation Python. Il s'agit d'une convention utilisée pour documenter des fonctions, des classes, des modules ou des méthodes en fournissant une description claire et concise de ce qu'ils font, de leurs paramètres, de leurs valeurs de retour, et de toute autre information pertinente. Les docstrings sont des commentaires spéciaux dans le code Python qui sont accessibles via l'attribut **doc** d'un objet.

Voici quelques points clés pour comprendre les docstrings en Python :

```
Format des Docstrings : Les docstrings sont généralement encadrées par trois guillemets simples
```

[5]:
```
def ma_fonction(parametre):
    """
    Description de la fonction.

    Args:
        parametre (type): Description du paramètre.

    Returns:
        type_de_retour: Description de la valeur de retour.
    """
    # Le code de la fonction
```

Contenu des Docstrings : Les docstrings doivent contenir des informations claires et utiles sur l'objet qu'ils documentent. Cela peut inclure :

```
Une brève description de ce que fait la fonction.
Une liste des paramètres, avec leur type et une description.
Une description de la valeur de retour de la fonction.
Des exemples d'utilisation.
```

Accès aux Docstrings : Vous pouvez accéder au contenu d'un docstring à l'aide de l'attribut **doc** de l'objet. Par exemple :

[6]:
```
print(ma_fonction.__doc__)
```

```
    Description de la fonction.

    Args:
        parametre (type): Description du paramètre.

    Returns:
```

```
        type_de_retour: Description de la valeur de retour.
```

Cela affichera la documentation de la fonction ma_fonction à la console.

Utilité des Docstrings : Les docstrings sont utiles pour plusieurs raisons :

```
Ils aident les autres développeurs (ou vous-même) à comprendre rapidement comment utiliser une f
Ils servent de référence lors de la documentation du code.
Ils sont utilisés par des outils de documentation automatique tels que Sphinx pour générer des d
```

1.3 Structures de Contrôle en Programmation

Les structures de contrôle jouent un rôle central dans la programmation en Python (et dans de nombreux autres langages). Elles sont essentielles pour définir le flux d'exécution d'un programme, permettant ainsi de prendre des décisions, de répéter des tâches et d'organiser le code de manière logique. Dans cette section, nous explorerons trois principales structures de contrôle qui sont fondamentales pour la création d'algorithmes efficaces :

- **Les Blocks d'Instructions :** Ces structures délimitent des sections de code qui s'exécutent ensemble, souvent à l'intérieur d'une fonction ou d'une boucle. Les blocs d'instructions contribuent à organiser et à structurer votre code de manière compréhensible.
- **Les Instructions Conditionnelles (If / Else) :** Les instructions conditionnelles permettent de prendre des décisions dans un programme. Selon une condition donnée, le programme peut exécuter différentes parties de code. Le constructeur "If" permet d'exécuter une action si une condition est vraie, tandis que "Else" permet de définir une action alternative si la condition n'est pas remplie.
- **Les Instructions Itératives (For, While) :** Les instructions itératives sont utilisées pour répéter des actions plusieurs fois. La boucle "For" permet de parcourir un ensemble de données ou de répéter un bloc de code un nombre spécifique de fois. La boucle "While" répète un bloc de code tant qu'une condition reste vraie.

1.3.1 Les Blocks d'instructions

En programmation, un bloc d'instructions, aussi appelé "suite d'instructions" ou "bloc de code", est une séquence d'instructions qui sont regroupées ensemble et exécutées comme une seule unité. Les blocs d'instructions sont utilisés pour organiser et structurer le code de manière logique et cohérente. Ils jouent un rôle essentiel pour rendre un programme lisible et maintenable. Voici quelques points clés pour comprendre les blocs d'instructions :

```
Définition des Blocs d'Instructions : Un bloc d'instructions est généralement délimité par des s
```

[ ]:
```python
def ma_fonction():
    # Ceci est le début du bloc d'instructions
    instruction_1
    instruction_2
    # ...
    instruction_n
    # Ceci est la fin du bloc d'instructions
```

Les blocs d'instructions servent principalement à :

- Regrouper des instructions associées : Les instructions à l'intérieur d'un bloc ont souvent un objectif commun ou sont liées à une tâche spécifique.
- Définir la portée : Les variables déclarées à l'intérieur d'un bloc sont généralement visibles uniquement à l'intérieur de ce bloc, ce qui permet de limiter leur portée (scope).
- Faciliter la lisibilité : En regroupant des instructions similaires dans un bloc, le code devient plus lisible et plus facile à comprendre.

Usage Courant : Les blocs d'instructions sont couramment utilisés dans les fonctions et les méthodes, où ils regroupent le code spécifique à la fonction. Les boucles et les structures conditionnelles utilisent également des blocs pour déterminer quelles parties du code doivent être répétées ou exécutées sous certaines conditions.

Emboîtement de Blocs : Il est important de noter que les blocs d'instructions peuvent être emboîtés, c'est-à-dire qu'un bloc peut contenir d'autres blocs. Cela permet de hiérarchiser et de structurer le code de manière complexe. Par exemple, une fonction peut contenir une boucle, et cette boucle peut elle-même contenir un bloc conditionnel, et ainsi de suite.

Indentation : En Python, contrairement à certains autres langages, l'indentation est cruciale pour définir les blocs d'instructions. Les lignes de code appartenant au même bloc doivent être indentées de la même manière, ce qui garantit la lisibilité du code

1.3.2 Les instructions conditionnelles :

Les instructions conditionnelles, également appelées structures de contrôle conditionnelles, sont un élément essentiel de la programmation. Elles permettent à un programme de prendre des décisions basées sur des conditions spécifiques. En Python, les instructions conditionnelles les plus couramment utilisées sont `if`, `elif` (else if), et `else`. Voici une explication détaillée des instructions conditionnelles :

1. **Instruction `if` :** L'instruction `if` permet d'exécuter un bloc de code uniquement si une condition spécifiée est évaluée comme vraie (True). La syntaxe de base est la suivante :

```
if condition:
    # Bloc de code à exécuter si la condition est vraie
```

Par exemple, vous pouvez utiliser une instruction if pour vérifier si un nombre est positif.

2. Instruction elif (Else If) : L'instruction elif est utilisée pour spécifier une condition alternative à vérifier si la première condition (if) n'est pas remplie. Elle permet de gérer plusieurs conditions différentes. Voici comment elle est utilisée :

```
if condition1:
    # Bloc de code à exécuter si condition1 est vraie
elif condition2:
    # Bloc de code à exécuter si condition2 est vraie
```

Par exemple, vous pouvez utiliser elif pour vérifier si un nombre est nul, positif ou négatif.

3. Instruction else : L'instruction else est utilisée pour spécifier un bloc de code à exécuter si aucune des conditions précédentes n'est vraie. Elle est souvent utilisée comme option de

secours lorsque toutes les autres conditions échouent. Voici comment elle est utilisée :

```
if condition:
    # Bloc de code à exécuter si la condition est vraie
else:
    # Bloc de code à exécuter si la condition n'est pas vraie
```

Par exemple, dans l'exemple précédent, le bloc de code associé à else sera exécuté lorsque nombre est égal à 0.

[1]:
```
def test_du_signe(valeur):
  if valeur < 0:
    print('négatif')
  elif valeur == 0:
    print('nul')
  else:
    print('positif')
```

[4]:
```
(test_du_signe(-2))
(test_du_signe(2))
(test_du_signe(0))
```

```
négatif
positif
nul
```

Note importante : Une condition est respectée si et seulement si elle correspond au résultat **booléen True**.

[5]:
```
valeur = -2
print(valeur < 0) # le résultat de cette comparaison est True

if valeur < 0:
  print('négatif')
```

```
True
négatif
```

Cela permet de développer des algorithmes avec des mélanges d'opéations Logiques et d'opérations de comparaisons. Par exemple : *si il fait beau et qu'il faut chaud, alors j'irai me baigner*

[6]:
```
x = 3
y = -1
if (x>0) and (y>0):
  print('x et y sont positifs')
else:
  print('x et y ne sont pas tous les 2 positifs')
```

```
x et y ne sont pas tous les 2 positifs
```

1.3.3 2. Les instructions itératives For, while

Dans Python, les instructions itératives, notamment les boucles for et while, sont des structures de contrôle permettant de répéter des actions ou des séquences d'instructions un certain nombre de fois ou tant qu'une condition est remplie. Elles sont essentielles pour automatiser des tâches répétitives et pour parcourir des collections de données telles que des listes, des chaînes de caractères, ou des dictionnaires. Voici une présentation et une description détaillée de chacune de ces boucles : #### 1- Boucle for a boucle for est utilisée pour parcourir une séquence (comme une liste, une chaîne de caractères, un dictionnaire, etc.) et exécuter un bloc de code pour chaque élément de la séquence. La syntaxe de base de la boucle for est la suivante :

```
for element in sequence:
    # Bloc de code à exécuter pour chaque élément
```

- element : Une variable qui prend la valeur de chaque élément de la séquence à chaque itération.
- sequence : La séquence à parcourir, telle qu'une liste, une chaîne de caractères, ou une plage générée par range(). Exemple d'utilisation de la boucle for :

[13]:
```
# range(début, fin, pas) est une built-in fonction tres utile de python qui␣
↪retourne un itérable.
for i in range(0, 10):
  print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[16]:
```
# on peut s'amuser a combiner cette boucle avec notre fonction de tout a l'heure.
for i in range(-10, 10, 2):
  print(i)
  test_du_signe(i)
```

```
-10
négatif
-8
négatif
-6
négatif
-4
négatif
-2
```

```
négatif
0
nul
2
positif
4
positif
6
positif
8
positif
```

1.3.4 2. Boucle While

La boucle while permet d'exécuter un bloc de code tant qu'une condition spécifiée reste vraie. Elle est utile lorsque vous ne savez pas à l'avance combien d'itérations seront nécessaires. La syntaxe de base de la boucle while est la suivante :

```
while condition:
    # Bloc de code à exécuter tant que la condition est vraie
```

condition : Une expression booléenne qui est évaluée à chaque itération. La boucle continue tant que cette condition est vraie.

[17]:
```
x = 0
while x < 10:
  print(x)
  x += 1 # incrémente x de 1 (équivalent de x = x+1)
```

```
0
1
2
3
4
5
6
7
8
9
```

1.4 Exercices

1.4.1 Exercices Partie 1

1. Modifiez la fonction e_potentielle définie plus haut pour retourner une valeur indiquant si l'energie calculée est supérieure ou inférieur a une **energie_limite** passée en tant que 4^{eme} argument
2. Créez une fonction Python qui prend deux nombres et un opérateur (+, -, *, /) en entrée. Effectuez l'opération arithmétique correspondante et renvoyez le résultat. Testez la fonction

avec diverses valeurs d'entrée et d'opérateurs.

3. Écrivez une fonction Python qui calcule l'aire d'un cercle en fonction de son rayon. Utilisez la formule `aire = π * r^2`, où π (pi) est une constante. Permettez à l'utilisateur de saisir le rayon et affichez l'aire.

4. Créez un programme Python capable de convertir entre les températures Celsius et Fahrenheit. Écrivez des fonctions pour la conversion de Celsius en Fahrenheit et vice versa. Demandez à l'utilisateur de saisir la température et affichez la température convertie.

5. Écrivez une fonction Python qui vérifie si un entier donné est pair ou impair. Utilisez des déclarations conditionnelles (if/else) pour déterminer et renvoyer le résultat. Testez la fonction avec divers nombres.

6. Développez une fonction Python qui vérifie si une chaîne donnée est un palindrome (se lit de la même manière de gauche à droite et de droite à gauche). Supprimez les espaces et convertissez la chaîne en minuscules pour une vérification insensible à la casse. Renvoyez True s'il s'agit d'un palindrome et False sinon. Testez la fonction avec différents mots et phrases.

1.4.2 Exercices Partie 2

1. Implémentez la **suite de Fibonacci** [0, 1, 1, 2, 3, 5, 8, 13, 21, ...] qui part de 2 nombres a=0 et b=1, et qui calcule le nombre suivant en additionnant les 2 nombres précédents.

Indices : - Pour cet exercice vous aurez besoin d'une bouce **While** - Vous pouvez imprimer cette suite jusqu'a atteindre un nombre **n** que vous aurez choisit - dans python il est possible de mettre a jour 2 variables simultannément sur la meme lignes : **a, b = b, a+b**

2. Implémentez l'algorithme du jeu "trouver le nombre" qui est le suivant : Un nombre aléatoire sera généré et le but est de le trouver.

- À chaque itération, l'utilisateur essaie de deviner le nombre.
- si l'utilisateur devine le numéro correctement, le jeu se terminera et un message de félicitations apparaîtra
- si l'utilisateur donne un nombre supérieur, un message "inférieur" apparaîtra
- si l'utilisateur donne un nombre bas, un message "supérieur" apparaîtra

[13]:
```
#vous pouvez utiliser la function random
import random
x= random.randint(0,1000)
print(x)
```

```
901
```

3. Write a Python program that prints all the multiples of 5 between 1 and 50 (inclusive).

4. Create a Python function that calculates the factorial of a given positive integer. Ask the user for input and display the result.

5. Create a Python program that generates prime numbers within a specified range. Ask the user to input the range and display the prime numbers found.

2 TP 2: Structure de données

2.1 Listes et Tuples

En Python, les listes et les tuples sont deux structures de données fondamentales qui permettent de stocker et de gérer des collections d'éléments. Bien qu'ils partagent certaines similitudes, ils présentent des différences essentielles en termes de mutabilité, d'utilisation et de syntaxe. Voici une introduction aux listes et aux tuples en Python :

- Listes en Python :

  Une liste est une collection ordonnée et mutable d'éléments. Les éléments d'une liste sont séparés par des virgules et sont entourés de crochets []. Les listes peuvent contenir des éléments de différents types (nombres, chaînes de caractères, autres listes, objets, etc.). Les listes peuvent être modifiées après leur création, ce qui signifie que vous pouvez ajouter, supprimer ou modifier des éléments. Pour accéder à un élément d'une liste, utilisez son index (l'indice), en commençant par 0 pour le premier élément. Exemple : ma_liste = [1, 2, 3, 'quatre', 'cinq']

- Tuples en Python :

  Un tuple est une collection ordonnée et immuable d'éléments. Les éléments d'un tuple sont séparés par des virgules et sont entourés de parenthèses (). Les tuples peuvent également contenir des éléments de différents types. Contrairement aux listes, les tuples ne peuvent pas être modifiés après leur création. Ils sont immuables. Pour accéder à un élément d'un tuple, utilisez son index de la même manière qu'avec les listes. Exemple : mon_tuple = (1, 2, 3, 'quatre', 'cinq')

- Utilisation Courante :

  Les listes sont couramment utilisées pour stocker des collections d'éléments lorsque vous avez besoin de modifier ces éléments au fil du temps. Par exemple, une liste peut représenter une liste de tâches à faire, des éléments dans un panier d'achat, etc. Les tuples sont souvent utilisés lorsque vous voulez vous assurer que les données restent constantes et ne doivent pas être modifiées accidentellement. Par exemple, les coordonnées géographiques d'un lieu, les dimensions d'un objet, etc.-

2.1.1 Création de Listes et de Tuples

une liste ou un tuple peuvent contenir tout types de valeurs (int, float, bool, string). On dit que ce sont des structures hétérogenes.

La différence entre les 2 est qu'une liste est **mutable** alors qu'un Tuple ne l'est pas (on ne peut pas le changer apres qu'il soit crée)

[1]:
```
# Listes
liste_1 = [1, 4, 2, 7, 35, 84]
villes = ['Paris', 'Berlin', 'Londres', 'Bruxelles']
nested_list = [liste_1, villes] # une liste peut meme contenir des listes ! On␣
 ↪appelle cela une nested list

#Tuples
tuple_1 = (1, 2, 6, 2)
```

[2]:
```
print(villes)
```

```
['Paris', 'Berlin', 'Londres', 'Bruxelles']
```

2.1.2 Indexing et Slicing

Indexing :

En Python, l'indexing fait référence à l'accès à un élément spécifique dans une séquence (comme une liste, une chaîne de caractères ou un tuple) en utilisant un numéro d'index. L'index est un entier qui identifie la position de l'élément dans la séquence. Voici quelques points importants concernant l'indexing :

- L'indexing commence généralement à 0 pour le premier élément de la séquence. Par exemple, 0 est l'index du premier élément, 1 est l'index du deuxième élément, et ainsi de suite.
- Vous pouvez également utiliser des indices négatifs pour compter à partir de la fin de la séquence. Par exemple, - 1 représente le dernier élément, - 2 le deuxième élément en partant de la fin, et ainsi de suite.
- Pour accéder à un élément spécifique, utilisez la notation [indice] après le nom de la séquence. Par exemple, ma_liste[0] accède au premier élément d'une liste.
- Si l'indice spécifié est en dehors de la plage valide, une erreur "IndexError" sera levée.

Slicing :

Le slicing en Python consiste à extraire une partie d'une séquence en spécifiant une plage d'indices. Il vous permet de créer une nouvelle séquence à partir d'une partie de la séquence d'origine. Voici comment fonctionne le slicing :

- Utilisez la notation [début:fin] après le nom de la séquence pour spécifier la plage d'indices que vous souhaitez extraire.
- Le slicing inclut l'élément au niveau de l'index de début, mais exclut l'élément au niveau de l'index de fin. Cela signifie que ma_liste[1:4] extrait les éléments à l'indice 1, 2 et 3, mais pas l'élément à l'indice 4.
- Si vous omettez l'indice de début, le slicing commencera à partir du premier élément. Si vous omettez l'indice de fin, le slicing ira jusqu'au dernier élément inclus.
- Vous pouvez également spécifier un pas en ajoutant un troisième paramètre [début:fin:pas]. Par exemple, ma_liste[0:5:2] extraira les éléments aux indices 0, 2 et 4.
- Le slicing fonctionne de la même manière pour les chaînes de caractères et les tuples.

[7]:
```
# INDEXING

print('séquence complete:', villes)
print('index 0:', villes[0])
print('index 1:', villes[1])
print('dernier index (-1):', villes[-1])
```

```
séquence complete: ['Paris', 'Berlin', 'Londres', 'Bruxelles']
index 0: Paris
index 1: Berlin
dernier index (-1): Bruxelles
```

[9]:
```
# SLICING [début (inclus) : fin (exclus) : pas]

print('séquence complete:', villes)
print('index 0-2:', villes[0:3])
print('index 1-2:', villes[1:3])
print('ordre inverse:', villes[::-1])
```

```
séquence complete: ['Paris', 'Berlin', 'Londres', 'Bruxelles']
index 0-2: ['Paris', 'Berlin', 'Londres']
index 1-2: ['Berlin', 'Londres']
ordre inverse: ['Bruxelles', 'Londres', 'Berlin', 'Paris']
```

2.1.3 Actions utiles sur les listes

- append(element): append is a list method that adds a specified element to the end of the list. It is commonly used when you want to add a single element to the existing list.
- insert(index, element): insert is a list method that allows you to insert an element at a specific index (position) within the list. You provide both the index where you want to insert the element and the element itself.
- extend(iterable): extend is a list method used to add multiple elements to the end of an existing list. It takes an iterable (such as a list, tuple, or string) as an argument and appends all its elements to the end of the list.
- sort(reverse=False): sort is a list method that arranges the elements of the list in a specific order. By default, it sorts the list in ascending order (from lowest to highest). You can also use reverse=True to sort in descending order.
- count(element): count is a list method that counts the number of occurrences of a specific element in the list. It is useful when you want to determine how many times a particular value appears in the list.

[3]:
```
villes = ['Paris', 'Berlin', 'Londres', 'Bruxelles'] # liste initiale
print(villes)

villes.append('Dublin') # Rajoute un élément a la fin de la liste
print(villes)

villes.insert(2, 'Madrid') # Rajoute un élément a l'index indiqué
print(villes)

villes.extend(['Amsterdam', 'Rome']) # Rajoute une liste a la fin de notre liste
print(villes)

print('longeur de la liste:', len(villes)) #affiche la longueur de la liste

villes.sort(reverse=False) # trie la liste par ordre alphabétique / numérique
print(villes)
```

```
print(villes.count('Paris')) # compte le nombre de fois qu'un élément apparait␣
↪dans la liste
```

```
['Paris', 'Berlin', 'Londres', 'Bruxelles']
['Paris', 'Berlin', 'Londres', 'Bruxelles', 'Dublin']
['Paris', 'Berlin', 'Madrid', 'Londres', 'Bruxelles', 'Dublin']
['Paris', 'Berlin', 'Madrid', 'Londres', 'Bruxelles', 'Dublin', 'Amsterdam',
'Rome']
longeur de la liste: 8
['Amsterdam', 'Berlin', 'Bruxelles', 'Dublin', 'Londres', 'Madrid', 'Paris',
'Rome']
1
```

Les listes et les tuples fonctionnent en harmonies avec les structures de controle **if/else** et **For**

[23]:
```
if 'Paris' in villes:
  print('oui')
else:
  print('non')
```

```
oui
```

[24]:
```
for element in villes:
  print(element)
```

```
Amsterdam
Berlin
Bruxelles
Dublin
Londres
Madrid
Paris
Rome
```

2.1.4 enumerate en Python :

La fonction enumerate en Python est une fonction intégrée qui est utilisée pour énumérer les éléments d'une séquence, comme une liste ou une chaîne de caractères, tout en fournissant leur indice (position) dans la séquence. Elle est couramment utilisée dans les boucles pour accéder à la fois à l'élément et à son indice. Voici une explication en français :

```
enumerate est une fonction qui prend en entrée une séquence (comme une liste ou une chaîne de ca
L'indice commence généralement à 0 pour le premier élément de la séquence, puis s'incrémente à c
La fonction enumerate est utile lorsque vous avez besoin à la fois de la valeur de l'élément et
Elle permet d'améliorer la lisibilité du code en évitant d'utiliser une variable de compteur pou
```

[4]:
```
for index, element in enumerate(villes):
  print(index, element)
```

```
0 Amsterdam
1 Berlin
2 Bruxelles
3 Dublin
4 Londres
5 Madrid
6 Paris
7 Rome
```

2.1.5 zip en Python :

La fonction zip en Python est une fonction intégrée qui combine plusieurs séquences (telles que des listes ou des tuples) en une seule séquence, en associant les éléments correspondants par position. Voici une explication en français :

```
zip prend en entrée une ou plusieurs séquences de même longueur et renvoie un objet itérable qui
Les séquences doivent avoir la même longueur, sinon zip s'arrêtera lorsque la séquence la plus c
La fonction zip est couramment utilisée pour regrouper des données connexes à partir de différen
Elle permet de simplifier le code en évitant l'utilisation de boucles pour parcourir simultanéme
```

[7]:
```
liste_2 = [312, 52, 654, 23, 65, 12, 678]
liste_3 = [1, 2, 3, 4, 5, 6, 7]
for element_1, element_2,element_3 in zip(villes, liste_2,liste_3):
  print(element_3," - ",element_1, element_2)
```

```
1  -  Amsterdam 312
2  -  Berlin 52
3  -  Bruxelles 654
4  -  Dublin 23
5  -  Londres 65
6  -  Madrid 12
7  -  Paris 678
```

2.2 Dictionnaires

Les dictionnaires sont des structures de controle **non-ordonnées**, c'est-a-dire que les valeurs qu'ils contiennent ne sont pas rangée selon un index, mais suivant une **clef unique**.

Une utilisation parfaite des dictionnaires est pour regrouper ensemble des "variables" dans un meme conténaire. (ces variables ne sont pas de vraies variables, mais des **keys**).

On peut par exemple crée un dictionnaire inventaire qui regroupe plusieurs produits (les clefs) et leur quantités (les valeurs)

1. Dict.values() :

   Dict.values() est une méthode de dictionnaire qui renvoie une vue d'ensemble (view) des valeurs du dictionnaire. Cette vue d'ensemble contient toutes les valeurs du dictionnaire, sans les clés associées. Il est couramment utilisé pour accéder à toutes les valeurs d'un dictionnaire sans avoir besoin de parcourir les clés.

2. Dict.keys() :

Dict.keys() est une méthode de dictionnaire qui renvoie une vue d'ensemble des clés du dictionnaire. Cette vue d'ensemble contient toutes les clés du dictionnaire, sans les valeurs associées. Elle est utilisée pour accéder à toutes les clés du dictionnaire, ce qui est utile lorsque vous avez besoin de parcourir les clés ou de vérifier si une clé spécifique existe dans le dictionnaire.

3. Dict.fromkeys() :

   Dict.fromkeys(iterable, valeur_par_défaut) est une méthode de classe qui crée un nouveau dictionnaire avec des clés à partir des éléments de l'itérable fourni. Chaque clé est associée à la même valeur par défaut spécifiée. Si aucune valeur par défaut n'est fournie, les clés sont associées à None. Elle est utilisée pour initialiser un dictionnaire avec des clés à partir d'une séquence donnée, par exemple une liste ou un tuple.

4. Obtenir des informations (inventaire['clé'] = 30 ou inventaire.get('pommes')) :

   Vous pouvez obtenir des informations à partir d'un dictionnaire en utilisant la notation de crochets (dictionnaire['clé']) pour accéder à la valeur associée à une clé spécifique. L'opération dictionnaire.get('clé') permet également d'obtenir la valeur associée à une clé, mais elle est plus sûre car elle ne génère pas d'erreur si la clé n'existe pas dans le dictionnaire. Elle renvoie plutôt None ou une valeur par défaut spécifiée.

5. Dict.pop("clé") :

   Dict.pop("clé") est une méthode qui supprime l'élément associé à la clé spécifiée dans le dictionnaire et renvoie la valeur de cet élément. Si la clé n'existe pas dans le dictionnaire, elle peut renvoyer une erreur si une valeur par défaut n'est pas spécifiée.

6. Dict.keys(), Dict.values(), Dict.items() :

   Dict.keys() renvoie une vue d'ensemble des clés du dictionnaire. Dict.values() renvoie une vue d'ensemble des valeurs du dictionnaire. Dict.items() renvoie une vue d'ensemble des paires clé-valeur du dictionnaire. Ces méthodes sont couramment utilisées pour parcourir ou examiner les clés, les valeurs ou les paires clé-valeur dans un dictionnaire.

[1]:
```
inventaire = {'pommes': 100,
              'bananes': 80,
              'poires': 120}
```

[2]:
```
inventaire.values()
```

[2]:
```
dict_values([100, 80, 120])
```

[3]:
```
inventaire.keys()
```

[3]:
```
dict_keys(['pommes', 'bananes', 'poires'])
```

[4]:
```
len(inventaire)
```

[4]:
```
3
```

Voici comment ajouter une association key/value dans notre dictionnaire (attention si la clef existe déja elle est remplacée)

[2]:
```
inventaire['abricots'] = 30
print(inventaire)
```

```
{'pommes': 100, 'bananes': 80, 'poires': 120, 'abricots': 30}
```

Attention : si vous cherchez une clef qui n'existe pas dans un dictionnaire, python vous retourne une erreur. Pour éviter cela, vous pouvez utiliser la méthode **get()**

[6]:
```
inventaire.get('peches') # n'existe pas
```

[7]:
```
inventaire.get('pommes') # pomme existe
```

[7]:
```
100
```

[22]:
```
Animal_list =('dog','cat','bird','fish')
Animal_number = 0
MyAnimals={

}
MyAnimals.fromkeys(Animal_list,Animal_number)
```

[22]:
```
{'dog': 0, 'cat': 0, 'bird': 0, 'fish': 0}
```

la méthode **pop()** permet de retirer une clef d'un dictionnaire tout en retournant la valeur associée a la clef.

[8]:
```
abricots = inventaire.pop("abricots")
print(inventaire) # ne contient plus de clef abricots
print(abricots) # abricots contient la valeur du dictionnaire
```

```
{'pommes': 100, 'bananes': 80, 'poires': 120}
30
```

Pour utiliser une boucle for avec un dictionnaire, il est utile d'utiliser la méthode **items()** qui retourne a la fois les clefs et les valeurs

[5]:
```
for key in inventaire.keys():
  print(f'this is the key: {key}')
print('--------------')
for value in inventaire.values():
  print('this is the values: '.format(value))
print('--------------')
for key, value in inventaire.items():
  print(f'this is key-value: {key},{value}' )
```

```
this is the key: pommes
this is the key: bananes
this is the key: poires
this is the key: abricots
--------------
```

```
this is the values:
this is the values:
this is the values:
this is the values:
--------------
this is key-value: pommes,100
this is key-value: bananes,80
this is key-value: poires,120
this is key-value: abricots,30
```

2.3 Les ensembles (Sets)

En Python, un ensemble (ou "set" en anglais) est une collection non ordonnée d'éléments uniques. Contrairement à d'autres structures de données comme les listes et les tuples, les ensembles ne permettent pas de stocker des éléments en doublon. Voici une explication détaillée sur les ensembles en Python :

1. Collection Non Ordonnée : Les ensembles ne maintiennent pas l'ordre d'insertion des éléments. Par conséquent, il n'y a pas d'indice associé à chaque élément dans un ensemble. L'ordre des éléments dans un ensemble peut varier lorsque vous parcourez l'ensemble.

2. Éléments Uniques : Les ensembles ne permettent pas de stocker plusieurs fois le même élément. Si vous tentez d'ajouter un élément déjà présent dans l'ensemble, il ne sera pas ajouté une seconde fois.

3. Définition d'un Ensemble : Pour créer un ensemble en Python, vous pouvez utiliser des accolades {} ou la fonction set(). Par exemple : `python mon_set = {1, 2, 3}` ou `mon_set = set([1, 2, 3])`.

4. Opérations sur les Ensembles : Les ensembles prennent en charge un ensemble d'opérations courantes, notamment l'ajout d'éléments, la suppression d'éléments, la vérification de l'appartenance d'un élément, l'union de deux ensembles, l'intersection, la différence, etc.

5. Utilisations Courantes : Les ensembles sont couramment utilisés pour effectuer des opérations ensemblistes, telles que la recherche d'éléments uniques dans une liste, la déduplication de données, la vérification de l'appartenance d'un élément à un ensemble de données, etc.

6. Immuabilité des Éléments : Les éléments stockés dans un ensemble doivent être immuables, c'est-à-dire qu'ils ne peuvent pas être modifiés une fois qu'ils sont ajoutés à l'ensemble. Les types de données immuables courants comprennent les nombres, les chaînes de caractères et les tuples.

7. Itérabilité : Vous pouvez parcourir les éléments d'un ensemble à l'aide d'une boucle for, mais rappelez-vous que l'ordre des éléments n'est pas garanti.

8. Méthodes utiles : Les ensembles disposent de nombreuses méthodes utiles, telles que add() pour ajouter un élément, remove() pour supprimer un élément, union() pour l'union de deux ensembles, intersection() pour l'intersection, difference() pour la différence, etc.

2.4 Exercice

1. Transformer le code suivant qui donne la **suite de Fibonacci** pour enregistrer les résultats dans une liste et retourner cette liste a la fin de la fonction

```
# Exercice :
def fibonacci(n):
    a = 0
    b = 1
    while b < n:
      a, b = b, a+b
      print(a)
```

2. Implémentez une fonction *trier(classeur, valeur)* qui place une valeur dans un dictionnaire en fonction de son signe

```
classeur = {'négatifs':[],
            'positifs':[]
            }
```

3. Implémentez un code qui permet de lire les valeur du classeur parameters

```
parameters = {'Ws':[1,2,3,4,8],
            'Bs':[1,2,3,5,6]
            }
```

4. Écrivez un programme Python qui permet à l'utilisateur de saisir une liste de nombres, puis ajoute un nombre supplémentaire à la fin de la liste. Ensuite, le programme doit compter combien de fois ce nombre supplémentaire apparaît dans la liste et afficher le résultat.

5. Créez un tuple contenant une série de nombres. Écrivez un programme Python qui génère un nouveau tuple contenant la somme cumulative des nombres du tuple d'origine. Par exemple, si le tuple initial est (1, 2, 3, 4), le nouveau tuple devrait être (1, 3, 6, 10).

6. Créez un dictionnaire de contacts en utilisant les noms comme clés et les numéros de téléphone comme valeurs. Écrivez un programme Python qui permet à l'utilisateur d'ajouter de nouveaux contacts, de rechercher un numéro de téléphone par nom, de supprimer un contact existant et d'afficher la liste complète des contacts.

7. Écrivez un programme Python qui prend une liste de nombres et supprime tous les éléments de la liste qui sont plus petits qu'une valeur seuil spécifiée par l'utilisateur. Affichez la liste résultante après la suppression.

8. Créez deux tuples contenant des noms d'étudiants. Écrivez un programme Python qui fusionne ces deux tuples en un seul, puis trie le tuple résultant par ordre alphabétique des noms. Affichez le tuple trié.

3 TP 3: List Comprehension et fonctions intégrées

3.1 List Comprehension

La compréhension de liste, souvent appelée "list comprehension" en anglais, est une manière concise et puissante de créer des listes en Python. Elle permet de générer rapidement une nouvelle liste

en appliquant une expression à chaque élément d'une séquence (comme une liste, un tuple, ou une chaîne de caractères) ou en utilisant des conditions pour filtrer les éléments de la séquence d'origine. Voici une explication détaillée de la compréhension de liste :

Syntaxe de base de la compréhension de liste :

La syntaxe générale de la compréhension de liste est la suivante :

```
nouvelle_liste = [expression for élément in séquence if condition]
```

- nouvelle_liste : C'est la nouvelle liste que vous allez créer.
- expression : C'est une expression Python qui sera évaluée pour chaque élément de la séquence.
- élément : C'est une variable temporaire qui prend la valeur de chaque élément de la séquence à chaque itération.
- séquence : C'est la séquence d'origine (liste, tuple, chaîne de caractères, etc.) que vous parcourez.
- condition (optionnelle) : C'est une condition qui filtre les éléments de la séquence. Seuls les éléments pour lesquels cette condition est vraie seront inclus dans la nouvelle liste.

Exemple d'utilisation :

Voici un exemple simple de compréhension de liste pour créer une liste des carrés des nombres de 0 à 9 :

```
carres = [x**2 for x in range(10)]
```

Dans cet exemple, x prend la valeur de chaque nombre de 0 à 9, l'expression x**2 calcule le carré de chaque nombre, et la nouvelle liste carres contient les carrés des nombres.

Utilisation de la condition :

Vous pouvez également utiliser une condition pour filtrer les éléments de la séquence d'origine. Par exemple, pour créer une liste des nombres pairs entre 0 et 9 :

```
nombres_pairs = [x for x in range(10) if x % 2 == 0]
```

Dans cet exemple, seuls les nombres pairs (ceux pour lesquels x % 2 == 0 est vrai) sont inclus dans la nouvelle liste nombres_pairs.

Avantages de la compréhension de liste :

- Concision : La compréhension de liste permet d'exprimer des opérations de création de liste de manière concise et lisible.
- Performance : Elle est généralement plus rapide que l'utilisation de boucles for traditionnelles pour la création de listes.
- Clarté : Elle améliore la lisibilité du code en évitant d'avoir à déclarer une liste vide et à ajouter des éléments un par un.

Les deux code ci-dessous effectuent chacun la meme opération. On peut voir (grace a la commande %%time) que le temps d'execution avec liste comprehenion est bien inférieur au temps d'execution avec la méthode append()

[5]:
```
%%time
liste = []
```

```
for i in range(1000000):
  liste.append(i**2)
```

```
CPU times: user 406 ms, sys: 22.3 ms, total: 428 ms
Wall time: 526 ms
```

[6]:
```
%%time
liste = [i**2 for i in range(1000000)]
```

```
CPU times: user 334 ms, sys: 27.7 ms, total: 361 ms
Wall time: 427 ms
```

[7]:
```
liste = [[i**2 for i in range(0,10,2)] for i in range(3)]
print(liste)
```

```
[[0, 4, 16, 36, 64], [0, 4, 16, 36, 64], [0, 4, 16, 36, 64]]
```

On peut rajouter des conditions **if** dans les listes comprehension, par exemple :

[8]:
```
liste = [i**2 for i in range(100000) if (i % 2) == 0] # calcule i**2 seulement␣
↪pour les nombres pairs.

print(liste[:10]) #affiche les 10 premiers éléments de la liste
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

[9]:
```
names=['adel','omar','moktar','ali']
dico1={name:idx for idx,name in enumerate(names)}
print((dico1))
print(dico1.keys())
print(dico1.values())
```

```
{'adel': 0, 'omar': 1, 'moktar': 2, 'ali': 3}
dict_keys(['adel', 'omar', 'moktar', 'ali'])
dict_values([0, 1, 2, 3])
```

[10]:
```
ages=[30,23,25,26]
dico2={names:ages for names,ages in zip(names,ages)}
print((dico2))
print(dico2.keys())
print(dico2.values())
```

```
{'adel': 30, 'omar': 23, 'moktar': 25, 'ali': 26}
dict_keys(['adel', 'omar', 'moktar', 'ali'])
dict_values([30, 23, 25, 26])
```

[16]:
```
ages=[30,23,25,26]
dico2={names:ages for names,ages in zip(names,ages) if ages< 30}
print((dico2))
```

```
print(dico2.keys())
print(dico2.values())
```

```
{'omar': 23, 'moktar': 25, 'ali': 26}
dict_keys(['omar', 'moktar', 'ali'])
dict_values([23, 25, 26])
```

3.2 Built-in Functions

Python contient un grand nombre de fonctions intégrées tres utiles a connaitre. Cela vous permet de construire des codes plus rapidement, sans avoir a développer vos propres fonctions pour les taches les plus basique. Dans ce notebook, je vous montre les plus importantes :

3.2.1 Fonction de bases

Utiles en toute circonstance !

[2]:
```
import numpy as np
x = np.pi
print(abs(x)) # valeur absolue
print(round(x,8)) # arrondi
```

```
3.141592653589793
3.14159265
```

[3]:
```
liste = [-2, 3, 1, 0, -4]

print(min(liste)) # minimum
print(max(liste)) # maximum
print(len(liste)) # longueur
print(sum(liste)) # somme des éléments
```

```
-4
3
5
-2
```

[4]:
```
liste = [False, False, True]

print(any(liste)) # y-a-t'il au moins un élément True ?
print(all(liste)) # est-ce-que tous les éléments sont True ?
```

```
True
False
```

3.2.2 Fonction de conversion

Il peut etre tres utile de convertir une variable d'un type a un autre (par exemple pour faire des calculs). Pour cela, on dispose des fonctions int(), str() et float().

La fonction **type()** est tres utile pour inspecter les types de nos variables

[5]:
```
age = '32'
type(age)
```

[5]:
```
str
```

[6]:
```
age = int(age)
type(age)
```

[6]:
```
int
```

[7]:
```
age + 10
```

[7]:
```
42
```

[8]:
```
float(x)
type(x)
```

[8]:
```
float
```

On peut également convertir des listes en tuples, ou des tableaux Numpy (que l'on verra par la suite) en liste...

[9]:
```
tuple_1 = (1, 2, 3, 4)

liste_1 = list(tuple_1) # convertir un tuple en liste
print(liste_1)
type(liste_1)
```

```
[1, 2, 3, 4]
```

[9]:
```
list
```

[10]:
```
print(bin(15),bin(32))
print(oct(15),oct(32))
print(hex(15),hex(32))
```

```
0b1111 0b100000
0o17 0o40
0xf 0x20
```

3.2.3 La fonction input()

La fonction input() en Python est une fonction intégrée qui permet à un programme d'obtenir des données en entrée depuis l'utilisateur via le clavier. Elle est principalement utilisée pour interagir avec l'utilisateur en demandant des informations ou des réponses à des questions. Voici une brève explication de la fonction input() :

- Lorsque la fonction input() est appelée, elle affiche un message d'invite (généralement une chaîne de caractères) à l'utilisateur, qui apparaît dans la console.
- L'utilisateur peut alors saisir des données depuis le clavier, suivi de la touche "Entrée".
- La fonction input() attend que l'utilisateur entre des données et retourne une chaîne de caractères (string) contenant les données saisies. L'application de la fonction input() est très utile pour créer des programmes interactifs, des questionnaires, des formulaires de saisie, etc.
- Il est important de noter que la fonction input() renvoie toujours une chaîne de caractères (string). Si vous avez besoin de manipuler des données numériques, vous devrez les convertir en types de données appropriés, tels que des entiers ou des nombres à virgule flottante, à l'aide des fonctions de conversion telles que int() ou float().

[12]:
```
age = input('quel age avez-vous ?')
```

```
quel age avez-vous ?30 ans
```

[14]:
```
type(age) # age est de type string. il faut penser a le convertir si on désire␣
↪faire un calcul avec
```

[14]:
```
str
```

3.2.4 La fonction format()

En Python, la méthode format() est utilisée pour formater des chaînes de caractères en y insérant des valeurs ou des variables dans des emplacements spécifiés, appelés "emplacements de remplacement" ou "placeholders". Elle permet de créer des chaînes de caractères dynamiques en insérant des valeurs dans des positions prédéfinies. Voici une explication en français :

- La méthode format() est appelée sur une chaîne de caractères (la chaîne de format) et accepte un ou plusieurs arguments qui seront insérés dans la chaîne aux emplacements spécifiés.
- Les emplacements de remplacement sont indiqués par des accolades {} dans la chaîne de format. Par exemple, "Bonjour, {} !" est une chaîne de format avec un emplacement de remplacement.
- Vous pouvez spécifier l'ordre dans lequel les valeurs seront insérées en utilisant des indices numériques à l'intérieur des accolades. Par exemple, "{} {} {}" utilisera les trois premiers arguments dans cet ordre.
- Vous pouvez également nommer les emplacements de remplacement pour une utilisation plus explicite. Par exemple, "Bonjour, {prenom} !" utilise un emplacement nommé {prenom}.
- Les valeurs à insérer sont passées comme arguments à la méthode format(), soit dans l'ordre, soit en utilisant des noms de paramètres si des emplacements nommés sont utilisés.
- La méthode format() effectue automatiquement la conversion de types, ce qui signifie que vous pouvez insérer des valeurs de différents types (entiers, nombres à virgule flottante, chaînes de caractères, etc.) dans la chaîne de format.

[15]:
```
x = 25
ville = 'Paris'

message = 'il fait {} degrés a {}'.format(x, ville)
print(message)
```

```
il faut 25 degrés a Paris
```

[17]:
```
message = f'il fait {x} degrées a {ville}'
print(message)
```

```
il fait 25 degrées a Paris
```

oici quelques astuces et techniques que vous pouvez utiliser avec la méthode format() en Python pour formater vos chaînes de caractères de manière plus efficace et lisible :

- Alignement du texte : Vous pouvez spécifier l'alignement du texte à l'intérieur de l'emplacement de remplacement en utilisant les caractères <, >, ou ^. Par exemple : "{:>10}".format("texte") alignera le texte à droite sur une largeur de 10 caractères.

[10]:
```
fruits=['banana','grapes','strawberries','pear','peach']
for fruit in fruits:
    print("-{:>15}".format(fruit))
```

```
-         banana
-         grapes
-    strawberries
-           pear
-          peach
```

- Remplacement conditionnel : Vous pouvez conditionnellement choisir quelle valeur insérer en fonction d'une condition. Par exemple : "Bonjour, {} !".format(nom if condition else "Cher utilisateur").

[12]:
```
a=3
condition=True
nom='Omar'
print("Bonjour, {} !".format(nom if condition else "Cher utilisateur"))

condition=False

print("Bonjour, {} !".format(nom if condition else "Cher utilisateur"))
```

```
Bonjour, Omar !
Bonjour, Cher utilisateur !
```

Répétition d'un motif : Vous pouvez répéter un motif dans un emplacement de remplacement. Par exemple : "-" * 20 crée une chaîne composée de 20 tirets.

[15]:
```
"-"*20,"*"*20
```

[15]:
```
('--------------------', '**********')
```

3.2.5 La fonction open()

Cette fonction est l'une des plus utile de Python. Elle permet d'ouvrir n'importe quel fichier de votre ordinateur et de l'utiliser dans Python. Différents modes existent : - le mode 'r' : lire un fichier

de votre ordinateur - le mode ‘w’ : écrire un fichier sur votre ordinateur - le mode ‘a’ : (append) ajouter du contenu dans un fichier existant

[5]:
```
f = open('./assets/text.txt', 'w') # ouverture d'un objet fichier f
f.write('hello')
f.close() # il faut fermer notre fichier une fois le travail terminé
```

[6]:
```
f = open('./assets/text.txt', 'r')
print(f.read())
f.close()
```

```
hello
```

Dans la pratique, on écrit plus souvent **with open() as f** pour ne pas avoir a fermer le fichier une fois le travail effectué :

[7]:
```
with open('./assets/text.txt', 'r') as f:
    print(f.read())
```

```
hello
```

3.3 Exercice

1. Le code ci-dessous permet de créer un fichier qui contient les nombres carrée de 0 jusqu’a 19. L’exercice est d’implémenter un code qui permet de lire ce fichier et d’écrire chaque ligne dans une liste.

Note_1 : la fonction **read().splitlines()** sera tres utile

Note_2 : Pour un meilleur résultat, essayer d’utiliser une liste comprehension !

[8]:
```
# Ce bout de code permet d'écrire le fichier
with open('./assets/fichier.txt', 'w') as f:
    for i in range(0, 20):
        f.write(f'{i}: {i**2} \n')
    f.close()

# Écrivez ici le code pour lire le fichier et enregistrer chaque lignes dans une␣
 ↪liste.
```

4 TP 4: Modules de Bases

Python contient un certain nombre de modules intégrés, qui offrent de nombreuses fonctions mathématiques, statistiques, aléatoires et os tres utiles. Pour importer un module, il faut procéder comme-ci dessous : - **import module** (importe tout le module) - **import module as md** (donne un surnom au module) - **from module import fonction** (importe une fonction du module)

[5]:
```
import math
import statistics
import random
```

```
import os
import glob
```

4.1 Modules math et statistics

les modules math et statistics sont en apparence tres utiles, mais en data science, nous utiliserons leurs équivalents dans le package **NUMPY**. Il peut néanmoins etre intéressant de voir les fonctions de bases.

[8]:
```
print(math.pi)
print(math.cos(2*math.pi))
print(math.exp(1))
```

```
3.141592653589793
1.0
2.718281828459045
```

[13]:
```
liste = [1, 4, 6, 2, 5, 3, 9, 6, 2, 1, 8, 8, 10, 9, 9]

print(statistics.mean(liste)) # moyenne de la liste
print(statistics.variance(liste)) # variance de la liste
print(statistics.median(liste)) # median de la liste
```

```
5.533333333333333
10.266666666666666
6
```

4.2 Module Random

Le module random est l'un des plus utile de Python. En datascience, nous utiliserons surtout sont équivalent **NUMPY**

[14]:
```
random.seed(0) # fixe le générateur aléatoire pour produire toujours le meme␣
↪résultat

print(random.choice(liste)) # choisit un élément au hasard dans la liste

print(random.random()) # génére un nombre aléatoire entre 0 et 1

print(random.randint(5, 10)) # génére un nombre entier aléatoire entre 5 et 10
```

```
9
0.3852453064766108
8
```

[15]:
```
random.sample(range(100), 10) # retourne une liste de 10 nombres aléatoires␣
↪entre 0 et 100
```

[15]:
```
[5, 33, 65, 62, 51, 38, 61, 45, 74, 27]
```

[16]:
```
print('liste de départ', liste)

random.shuffle(liste) #mélange les éléments d'une liste

print('liste mélangée', liste)
```

```
liste de départ [1, 4, 6, 2, 5, 3, 9, 6, 2, 1, 8, 8, 10, 9, 9]
liste mélangée [2, 9, 8, 9, 3, 1, 6, 8, 10, 1, 4, 9, 5, 6, 2]
```

4.3 Modules OS et Glob

4.3.1 Le module os :

Le module os en Python est une bibliothèque intégrée qui permet d'interagir avec le système d'exploitation sous-jacent, que ce soit Windows, macOS ou Linux. Il offre un ensemble de fonctions pour effectuer des opérations liées au système de fichiers, à la gestion des répertoires, à la manipulation des chemins, et bien plus encore. Voici quelques-unes des opérations courantes que vous pouvez effectuer avec le module os :

- Navigation dans le système de fichiers : Vous pouvez obtenir des informations sur le répertoire de travail actuel avec os.getcwd(). Pour changer de répertoire, vous pouvez utiliser os.chdir(chemin).
- Création et suppression de répertoires : Vous pouvez créer un nouveau répertoire avec os.mkdir(chemin) et un répertoire récursivement avec os.makedirs(chemin). Pour supprimer un répertoire, utilisez os.rmdir(chemin) et os.removedirs(chemin).
- Liste de fichiers et répertoires : Pour obtenir la liste des fichiers et répertoires dans un répertoire donné, utilisez os.listdir(chemin).
- Manipulation des chemins de fichiers : Le module os.path fournit des fonctions pour manipuler des chemins de fichiers de manière portable, comme os.path.join(), os.path.basename(), os.path.dirname(), etc.
- Exécution de commandes système : Vous pouvez exécuter des commandes système en utilisant os.system(commande).
- Interrogation des informations système : Le module os fournit des fonctions pour obtenir des informations sur le système, comme le nom de l'utilisateur actuel (os.getlogin()), le nom de l'ordinateur (os.uname() sur Unix), etc.

[17]:
```
os.getcwd() # affiche le répertoire de travail actuel
```

[17]:
```
'/Users/mac/Documents/Teaching/Machine learning/Python-Machine-
Learning/Formation python for machine learning/Introduction to python'
```

4.3.2 Le module glob :

Le module glob en Python permet de rechercher des fichiers et des répertoires en utilisant des motifs (patterns) basés sur des caractères génériques, similaires à ceux utilisés dans les expressions

régulières. Voici ce que vous pouvez faire avec le module glob :

- Recherche de fichiers et répertoires : Vous pouvez utiliser glob.glob(motif) pour rechercher des fichiers ou répertoires qui correspondent à un motif donné. Par exemple, glob.glob("*.txt") renverra une liste de tous les fichiers avec l'extension .txt dans le répertoire actuel.
- Recherche récursive : Pour effectuer une recherche récursive dans les sous-répertoires, vous pouvez utiliser glob.glob("dossier/**/*", recursive=True) (disponible à partir de Python 3.5).
- Utilisation de caractères génériques : Le module glob prend en charge des caractères génériques tels que * (correspond à n'importe quel nombre de caractères), ? (correspond à un seul caractère) et [...] (correspond à un ensemble de caractères).

Exemples d'utilisation : Le module glob est utile pour parcourir des fichiers dans un répertoire, effectuer des opérations sur des fichiers correspondant à un modèle donné, ou créer des listes de fichiers à traiter.

[18]:
```
print(glob.glob('*')) # contenu du repertoire de travail actuel
```

```
['Solutoin exercises.ipynb', '2-Structures de Controles.ipynb', '1-Variables et
Fonctions.ipynb', '7-Modules de Bases.ipynb', '8-Programmation Orientee
Objet.ipynb', '__pycache__', '4-Dictionnaires.ipynb', '3-Structures de donnees
(Listes et Tuples).ipynb', '6- Built-in Functions.ipynb', 'MyModule.py',
'assets', '5-List Comprehension.ipynb']
```

4.4 Créer notre propre module

Vous pouvez également créer vos propres modules et les importer dans d'autres projets. Un module n'est en fait qu'un simple fichier.py qui contient des fonctions et des classes

[1]:
```
import MyModule as a
```

[3]:
```
a.somme(4,5)
a.substraction(5,9)
```

[3]:
```
-4
```

[4]:
```
from MyModule import somme
a.somme(4,5)
```

[4]:
```
9
```

4.5 Exercices

- Calculer l'écart-type d'un ensemble de données volumineux de nombres en utilisant la bibliothèque statistics.
- Implémenter une fonction pour générer un mot de passe aléatoire qui inclut des lettres majuscules, des lettres minuscules, des chiffres et des caractères spéciaux, avec une longueur personnalisable.

- Écrire un programme qui recherche tous les fichiers Python (.py) dans un répertoire et ses sous-répertoires en utilisant les bibliothèques os et glob, puis compte le nombre total de lignes de code dans l'ensemble de ces fichiers.
- Créer un programme qui simule un jeu de dés simple. Lancez deux dés en utilisant la bibliothèque random, calculez la somme des lancers, et suivez le nombre de fois où chaque somme se produit au cours de plusieurs manches.
- Calculer la moyenne, la médiane et le mode d'un ensemble de données en utilisant la bibliothèque statistics, puis affichez les résultats dans un tableau formaté.
- Construire un système de gestion de fichiers qui permet aux utilisateurs de créer, supprimer et répertorier des répertoires et des fichiers dans un emplacement spécifié en utilisant la bibliothèque os. Implémentez la gestion d'erreurs pour divers scénarios.
- Développer un programme qui génère une liste de nombres premiers dans une plage spécifiée en utilisant une combinaison de la bibliothèque math pour les opérations mathématiques et une boucle pour vérifier la primalité.
- Créer une fonction qui calcule le plus grand commun diviseur (PGCD) d'une liste de nombres en utilisant la fonction gcd() de la bibliothèque math et la récursivité.
- Construire un jeu-questionnaire textuel où les questions sont lues à partir d'un fichier, les réponses sont mélangées, et les utilisateurs peuvent sélectionner les réponses. Suivez et affichez le score de l'utilisateur.
- Implémenter un programme qui simule un portefeuille d'actions de base. Les utilisateurs peuvent acheter et vendre des actions, voir la valeur actuelle de leur portefeuille, et suivre leurs gains/pertes au fil du temps. Utilisez des fichiers pour stocker les données historiques.

5 TP 5: Programmation Orientée Objet

Qu'est-ce que la Programmation Orientée Objet (POO) ? La Programmation Orientée Objet (POO) est un paradigme de programmation qui repose sur la notion d'objets. Les objets sont des entités autonomes qui regroupent des données (attributs) et des fonctionnalités (méthodes) associées. En POO, le code est organisé autour d'objets qui interagissent entre eux. La POO vise à modéliser le monde réel en utilisant des objets pour représenter des entités et des actions.

[1]:
```python
# Exemple simple d'objet en Python
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def se_presenter(self):
        print(f"Je m'appelle {self.nom} et j'ai {self.age} ans.")

# Création d'une instance de la classe Personne
personne1 = Personne("Alice", 30)
personne1.se_presenter()
```

```
Je m'appelle Alice et j'ai 30 ans.
```

5.1 Avantages de la POO en Python

La POO présente plusieurs avantages en Python, notamment :

- Modularité : Les objets permettent de regrouper le code en modules réutilisables.
- Encapsulation : Les données et les fonctionnalités sont encapsulées dans des objets, ce qui permet de les protéger et de les rendre plus sûres.
- Abstraction : La POO permet de masquer les détails complexes et de se concentrer sur l'interface de l'objet.
- Héritage : Les classes peuvent hériter de fonctionnalités d'autres classes, favorisant la réutilisation du code.
- Polymorphisme : Les objets peuvent être utilisés de manière polymorphe, c'est-à-dire que des objets de classes différentes peuvent répondre à la même interface.

[2]:
```python
# Exemple d'héritage en Python
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def parler(self):
        pass

class Chat(Animal):
    def parler(self):
        return "Miaou !"

class Chien(Animal):
    def parler(self):
        return "Wouaf !"

# Utilisation du polymorphisme
chat = Chat("Minou")
chien = Chien("Rex")

animaux = [chat, chien]
for animal in animaux:
    print(animal.nom + ": " + animal.parler())
```

```
Minou: Miaou !
Rex: Wouaf !
```

5.2 Objets et Classes

En POO, une classe est un modèle pour la création d'objets. Une classe définit la structure et le comportement des objets qui en sont issus. Les objets sont des instances de classes.

[3]:
```python
# Définition d'une classe en Python
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def afficher_details(self):
        print(f"Marque : {self.marque}, Modèle : {self.modele}")

# Création d'objets à partir de la classe Voiture
voiture1 = Voiture("Toyota", "Camry")
voiture2 = Voiture("Honda", "Civic")

# Appel d'une méthode de l'objet
voiture1.afficher_details()
voiture2.afficher_details()
```

```
Marque : Toyota, Modèle : Camry
Marque : Honda, Modèle : Civic
```

5.3 Attributs et Méthodes

Les attributs sont des variables qui stockent des données liées à un objet, tandis que les méthodes sont des fonctions associées à un objet. Les attributs et les méthodes définissent les caractéristiques et le comportement des objets.

[4]:
```python
# Exemple d'attributs et de méthodes en Python
class Etudiant:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
        self.notes = []

    def ajouter_note(self, note):
        self.notes.append(note)

    def moyenne_notes(self):
        if len(self.notes) == 0:
            return 0
        return sum(self.notes) / len(self.notes)

# Création d'un objet de la classe Etudiant
etudiant1 = Etudiant("Alice", 20)

# Utilisation des méthodes et des attributs
etudiant1.ajouter_note(85)
etudiant1.ajouter_note(92)
moyenne = etudiant1.moyenne_notes()
```

5.4 Encapsulation

L'encapsulation est le principe qui consiste à encapsuler (cacher) les détails internes d'un objet et à fournir une interface pour interagir avec lui. En Python, l'encapsulation est généralement réalisée en définissant des attributs comme publics, privés ou protégés en utilisant des conventions de nommage (par exemple, _attribut pour un attribut protégé).

[8]:
```python
# Exemple d'encapsulation en Python
class CompteBancaire:
    def __init__(self, solde):
        self._solde = solde

    def deposer(self, montant):
        if montant > 0:
            self._solde += montant

    def retirer(self, montant):
        if 0 < montant <= self._solde:
            self._solde -= montant

    def consulter_solde(self):
        return self._solde

# Création d'un objet de la classe CompteBancaire
compte = CompteBancaire(1000)

# Utilisation des méthodes pour effectuer des opérations sur le solde
compte.deposer(500)
compte.retirer(300)
solde_actuel = compte.consulter_solde()
solde_actuel
```

[8]:
```
1200
```

5.5 Définition d'une Classe

En programmation orientée objet (POO), une **classe** est un modèle ou un plan pour créer des objets. Elle définit la structure et le comportement des objets qui seront créés à partir d'elle. La définition d'une classe se fait en utilisant le mot-clé `class`.

```python
# Exemple de définition d'une classe en Python
class Voiture:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
```

Dans cet exemple, nous avons défini une classe `Voiture` avec deux attributs : `marque` et `modele`. La méthode `__init__` est un constructeur qui initialise les attributs de l'objet lorsqu'il est créé.

5.5.1 Constructeurs et Destructeurs

Un **constructeur** est une méthode spéciale d'une classe qui est appelée lorsqu'un nouvel objet de cette classe est créé. En Python, le constructeur est généralement appelé `__init__`.

```
# Exemple de constructeur en Python
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
```

Il existe également des méthodes spéciales telles que le **destructeur** `__del__`, qui est appelé lorsque l'objet est détruit. Cependant, le destructeur est rarement utilisé en Python car la gestion de la mémoire est gérée automatiquement.

5.5.2 Variables de Classe vs Variables d'Instance

Les **variables de classe** sont partagées par toutes les instances d'une classe, tandis que les **variables d'instance** sont spécifiques à chaque instance de la classe.

```
# Exemple de variables de classe et variables d'instance
class CompteBancaire:
    taux_interet = 0.05  # Variable de classe

    def __init__(self, solde):
        self.solde = solde  # Variable d'instance
```

Dans cet exemple, `taux_interet` est une variable de classe partagée par toutes les instances de `CompteBancaire`, tandis que `solde` est une variable d'instance spécifique à chaque compte bancaire.

5.6 Exercices

- Créer une Classe et des Instances : Définissez une classe Personne avec des attributs tels que nom, âge, et pays. Créez deux instances de la classe et affichez leurs détails.
- Héritage et Redéfinition de Méthode : Créez une classe de base Forme avec une méthode surface. Ensuite, créez deux sous-classes, Rectangle et Cercle, qui héritent de Forme et redéfinissent la méthode surface pour calculer leurs aires respectives.
- Encapsulation et Décorateurs de Propriété : Créez une classe CompteBancaire avec des attributs privés pour solde et numéro_de_compte. Utilisez les décorateurs de propriété pour permettre un accès contrôlé à ces attributs, autorisant les dépôts et les retraits tout en préservant l'encapsulation.
- Polymorphisme et Classes de Base Abstraites : Définissez une classe de base abstraite Animal avec une méthode parler(). Créez deux sous-classes, Chien et Chat, et implémentez la méthode parler() différemment dans chaque sous-classe. Démontrez le polymorphisme en appelant parler() sur des instances des deux classes.
- Variables de Classe et Méthodes de Classe : Créez une classe Étudiant avec une variable de classe total_étudiants pour suivre le nombre total d'étudiants. Implémentez une méthode de classe pour incrémenter ce compteur lorsqu'un nouvel objet étudiant est créé. Testez la classe en créant plusieurs instances d'étudiants et en affichant le nombre total d'étudiants.

6 TP 6: Numpy

NumPy est le package fondamental pour le calcul scientifique en Python. Il s'agit d'une bibliothèque Python qui fournit un objet tableau multidimensionnel, divers objets dérivés (tels que des tableaux masqués et des matrices) et un assortiment de routines pour des opérations rapides sur des tableaux, y compris mathématiques, logiques, manipulation de forme, tri, sélection, E/S , transformées de Fourier discrètes, algèbre linéaire de base, opérations statistiques de base, simulation aléatoire et bien plus encore.

Pour accéder à NumPy et à ses fonctions, importez-le dans votre code Python comme ceci :

[1]:
```
import numpy as np
```

Nous raccourcissons le nom importé en np pour une meilleure lisibilité du code à l'aide de NumPy. Il s'agit d'une convention largement adoptée que vous devez suivre afin que toute personne travaillant avec votre code puisse facilement la comprendre.

6.1 Générateurs de tableaux ndarray

La méthode la plus simple pour créer un tableau NumPy est de partir d'une liste Python existante. Vous pouvez utiliser la fonction `numpy.array()` pour convertir une liste en un tableau NumPy.

```
import numpy as np

# Création d'un tableau NumPy à partir d'une liste
ma_liste = [1, 2, 3, 4, 5]
mon_tableau = np.array(ma_liste)
```

6.1.1 Création de Tableaux NumPy avec des Valeurs Initiales

Vous pouvez également créer un tableau NumPy avec des valeurs initiales en utilisant des fonctions spécifiques de NumPy. Par exemple, la fonction `numpy.zeros()` crée un tableau rempli de zéros, tandis que `numpy.ones()` crée un tableau rempli de uns.

```
# Création d'un tableau rempli de zéros
tableau_zeros = np.zeros((3, 4))  # Un tableau de forme (3, 4)

# Création d'un tableau rempli de uns
tableau_ones = np.ones((2, 2))  # Un tableau de forme (2, 2)
```

6.1.2 Création de Séquences Numériques

NumPy propose des fonctions pour créer des séquences numériques régulières. Par exemple, vous pouvez utiliser `numpy.arange()` pour générer une séquence de nombres espacés de manière régulière.

```
# Création d'une séquence de nombres de 0 à 9
sequence = np.arange(10)  # Les nombres vont de 0 à 9
```

6.1.3 Génération de Nombres Aléatoires

Pour créer des tableaux NumPy avec des nombres aléatoires, NumPy propose le module `numpy.random`. Vous pouvez utiliser `numpy.random.rand()` pour générer des nombres aléatoires

dans une distribution uniforme entre 0 et 1.

```
# Génération de nombres aléatoires entre 0 et 1
nombres_aleatoires = np.random.rand(3, 3)  # Un tableau 3x3 de nombres aléatoires
```

[2]:
```
A = np.array([1, 2, 3]) # générateur par défaut, qui permet de convertir des␣
 ↪listes (ou autres objets) en tableau ndarray
A = np.zeros((2, 3)) # tableau de 0 aux dimensions 2x3
B = np.ones((2, 3)) # tableau de 1 aux dimensions 2x3
np.random.seed(0)
C = np.random.randn(2, 3) # tableau aléatoire (distribution normale) aux␣
 ↪dimensions 2x3
D = np.random.rand(2, 3) # tableau aléatoire (distribution uniforme)
size = (2, 3)
E = np.random.randint(0, 10, size) # tableau d'entiers aléatoires de 0 a 10 et␣
 ↪de dimension 2x3
B = np.eye(4, dtype=bool) # créer une matrice identité et convertit les éléments␣
 ↪en type bool.
```

[3]:
```
A = np.linspace(0,5, 20) # np.linspace(start,end, number of elements)
B = np.arange(0, 5, 0.5) # np.linspace(start,end, step)
print(A)
print(B)
```

```
[0.         0.26315789 0.52631579 0.78947368 1.05263158 1.31578947
 1.57894737 1.84210526 2.10526316 2.36842105 2.63157895 2.89473684
 3.15789474 3.42105263 3.68421053 3.94736842 4.21052632 4.47368421
 4.73684211 5.        ]
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5]
```

[7]:
```
A = np.linspace(5, 10,7, dtype=np.float16) # définit le type et la place a␣
 ↪occuper sur la mémoire
print("float16: ",A)
A = np.linspace(5,10,7, dtype=np.float32)
print("float32: ",A)
A = np.linspace(5,10,7, dtype=np.float64)
print("float64: ",A)
```

```
float16:  [ 5.     5.832  6.668  7.5    8.336  9.164 10.   ]
float32:  [ 5.         5.8333335  6.6666665  7.5        8.333333   9.166667
 10.       ]
float64:  [ 5.          5.83333333  6.66666667  7.5         8.33333333
9.16666667
 10.        ]
```

6.2 Attributs importants

[8]:
```
A = np.zeros((2, 3)) # création d'un tableau de shape (2, 3)

print(A.size) # le nombre d'éléments dans le tableau A
print(A.shape) # les dimensions du tableau A (sous forme de Tuple)
print(type(A.shape)) # voici la preuve que la shape est un tuple
print(A.shape[0]) # le nombre d'éléments dans la premiere dimension de A
```

```
6
(2, 3)
<class 'tuple'>
2
```

6.3 Méthodes importantes

6.3.1 reshape() : Redimensionner un Tableau

La fonction `reshape()` de NumPy permet de modifier la forme d'un tableau existant sans changer les données qu'il contient. Elle est utilisée pour transformer un tableau multi-dimensionnel en une forme différente tout en maintenant le même nombre total d'éléments.

```
# Création d'un tableau de forme (4, 3)
tableau_original = np.arange(12).reshape(4, 3)

# Redimensionnement en un tableau de forme (3, 4)
tableau_redimensionné = tableau_original.reshape(3, 4)
```

La fonction `reshape()` est utile lorsque vous avez besoin de changer la structure d'un tableau NumPy pour qu'il corresponde aux exigences d'une opération spécifique.

6.3.2 ravel() : Aplatir un Tableau

La fonction `ravel()` est utilisée pour aplatir un tableau multidimensionnel en un tableau à une seule dimension. Elle crée une vue du tableau original, ce qui signifie que les données ne sont pas copiées, mais elles sont simplement réorganisées pour former une seule dimension.

```
import numpy as np

# Création d'un tableau multidimensionnel
tableau_original = np.array([[1, 2, 3], [4, 5, 6]])

# Aplatir le tableau en un tableau à une dimension
tableau_aplati = tableau_original.ravel()
```

L'utilisation de `ravel()` est courante lorsque vous avez besoin de travailler avec des données sous forme de tableau à une seule dimension, par exemple lors de l'application de certaines opérations mathématiques.

6.3.3 squeeze() : Supprimer les Dimensions Unitaires

La fonction `squeeze()` est utilisée pour supprimer les dimensions qui ont une taille de 1 dans un tableau NumPy. Elle permet de simplifier la structure du tableau en éliminant les dimensions inutiles.

```
import numpy as np

# Création d'un tableau avec une dimension unitaire
tableau_original = np.array([[[1, 2, 3]]])

# Suppression de la dimension unitaire
tableau_squeezé = np.squeeze(tableau_original)
```

L'utilisation de `squeeze()` est utile lorsque vous avez des dimensions inutiles dans votre tableau et que vous souhaitez les éliminer pour simplifier le tableau.

6.3.4 concatenate() : Assembler des Tableaux

La fonction `concatenate()` de NumPy permet d'assembler deux tableaux le long d'un axe spécifié. Vous pouvez choisir l'axe le long duquel les tableaux seront concaténés (0 pour les lignes, 1 pour les colonnes, etc.).

```
import numpy as np

# Création de deux tableaux
tableau1 = np.array([[1, 2], [3, 4]])
tableau2 = np.array([[5, 6]])

# Concaténation le long de l'axe des colonnes (axis=1)
tableau_concaténé = np.concatenate((tableau1, tableau2.T), axis=1)
```

La fonction `concatenate()` est utile lorsque vous avez besoin de fusionner deux tableaux NumPy le long d'un axe spécifié pour former un nouveau tableau.

[9]:
```
A = np.zeros((2, 3)) # création d'un tableau de shape (2, 3)

A = A.reshape((3, 2)) # redimensionne le tableau A (3 lignes, 2 colonnes)
print("reshape: ",A)
B = A.ravel() # Aplatit le tableau A (une seule dimension)
print("ravel: ",B)
```

```
reshape:  [[0. 0.]
 [0. 0.]
 [0. 0.]]
ravel:  [0. 0. 0. 0. 0. 0.]
```

[10]:
```
A= np.array([1,2,5,3])
print(A)
print(A.shape)
```

```
A= A.reshape((A.shape[0],1))
print(A.shape)

A=A.squeeze()
print(A.shape)
```

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

6.3.5 concatenate()

La fonction `concatenate()` en NumPy est une fonction polyvalente utilisée pour concaténer deux tableaux ou plus le long d'un axe spécifié. Vous pouvez spécifier l'axe le long duquel les tableaux doivent être concaténés. Par défaut, elle concatène le long de l'axe 0 (les lignes), créant un tableau plus grand.

```
import numpy as np

# Création de deux tableaux
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Concaténation le long de l'axe 0 (les lignes)
result = np.concatenate((arr1, arr2))
```

Dans cet exemple, `result` sera `[1, 2, 3, 4, 5, 6]`.

Vous pouvez également spécifier explicitement l'axe, comme ceci :

```
result = np.concatenate((arr1, arr2), axis=0)
```

6.3.6 vstack()

La fonction `vstack()` est une fonction spécialisée pour l'empilement vertical des tableaux, ce qui signifie qu'elle empile les tableaux les uns au-dessus des autres le long de l'axe 0 (les lignes). Elle est équivalente à l'utilisation de `concatenate()` avec `axis=0`.

```
import numpy as np

# Création de deux tableaux
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6]])

# Empilement vertical des tableaux
result = np.vstack((arr1, arr2))
```

Dans cet exemple, `result` sera :

```
array([[1, 2],
       [3, 4],
```

```
       [5, 6]])
```

6.3.7 hstack()

La fonction `hstack()` est utilisée pour l'empilement horizontal des tableaux, ce qui signifie qu'elle empile les tableaux côte à côte le long de l'axe 1 (les colonnes). Elle est équivalente à l'utilisation de `concatenate()` avec `axis=1`.

```
import numpy as np

# Création de deux tableaux
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5], [6]])

# Empilement horizontal des tableaux
result = np.hstack((arr1, arr2))
```

Dans cet exemple, `result` sera :

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

`vstack()` et `hstack()` sont souvent utilisées lorsque vous avez des tableaux avec des formes compatibles et que vous souhaitez les combiner dans une direction spécifique sans spécifier explicitement l'axe.

[11]:
```
A = np.zeros((2, 3)) # création d'un tableau de shape (2, 3)
B = np.ones((2, 3)) # création d'un tableau de shape (2, 3)

np.concatenate((A, B), axis=0) # axe 0 : équivalent de np.vstack((A, B))
```

[11]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

[12]:
```
np.concatenate((A, B), axis=1) # axe 1 : équivalent de np.hstack((A, B))
```

[12]:
```
array([[0., 0., 0., 1., 1., 1.],
       [0., 0., 0., 1., 1., 1.]])
```

[8]:
```
A = np.zeros((2, 3)) # création d'un tableau de shape (2, 3)
B = np.ones((2, 3)) # création d'un tableau de shape (2, 3)
C = np.hstack((A,B))
print(C)
print(C.shape)
```

```
[[0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]]
(2, 6)
```

[9]:
```
C = np.vstack((A,B))
print(C)
print(C.shape)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [1. 1. 1.]
 [1. 1. 1.]]
(4, 3)
```

6.4 Slicing et Indexing

6.4.1 Indexing (Indexation)

L'indexation en NumPy consiste à accéder à des éléments spécifiques d'un tableau en utilisant des indices. Les indices sont utilisés pour cibler des éléments individuels ou des sous-tableaux en fonction de leur position dans le tableau.

Par exemple, pour accéder à un élément spécifique d'un tableau, vous utilisez un ou plusieurs indices entre crochets :

```
import numpy as np

# Création d'un tableau NumPy
tableau = np.array([1, 2, 3, 4, 5])

# Accès à un élément individuel par son index
element = tableau[2]  # Accède à l'élément à l'index 2 (valeur 3)
```

L'indexation en NumPy commence toujours à 0, donc l'élément à l'index 2 est le troisième élément du tableau.

6.4.2 Slicing (Découpage)

Le "slicing" en NumPy permet d'extraire des parties spécifiques d'un tableau en spécifiant une plage d'indices. Il est utilisé pour obtenir des sous-tableaux à partir d'un tableau plus grand.

Le "slicing" se fait en utilisant la notation avec deux points `:` entre les indices de début et de fin. Par exemple, pour extraire une partie d'un tableau :

```
import numpy as np

# Création d'un tableau NumPy
tableau = np.array([1, 2, 3, 4, 5])

# Extraction d'une sous-partie du tableau
sous_tableau = tableau[1:4]  # Extrait les éléments de l'index 1 à 3 inclus (valeurs 2, 3, 4)
```

Dans cet exemple, `sous_tableau` contiendra `[2, 3, 4]`. Le premier indice est inclus, tandis que le second indice est exclu.

Le “slicing” en NumPy peut également être utilisé avec des tableaux multidimensionnels pour extraire des sous-tableaux de manière similaire le long de plusieurs axes.

[ ]:
```
A = np.array([[1, 2, 3], [4, 5, 6]])
print(A)
```

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

[14]:
```
# Pour acceder a la ligne 0, colonne 1
A[0, 1]
```

[14]:
```
10
```

[16]:
```
# Pour selectionner les blocs de la ligne (0-1) colonne (0-1)
A[0:2, 0:2]
```

[16]:
```
array([[10, 10],
       [10, 10]])
```

[17]:
```
A[0:2, 0:2] = 10
print(A)
```

```
[[10 10  3]
 [10 10  6]]
```

6.5 Boolean Indexing

L’indexation booléenne est une technique puissante en NumPy qui permet de filtrer et d’extraire des éléments d’un tableau en se basant sur des conditions logiques. Cette méthode repose sur l’utilisation de tableaux de valeurs booléennes pour sélectionner les éléments qui satisfont une condition spécifique.

6.5.1 Création d’un Tableau Booléen

Pour effectuer une indexation booléenne, vous commencez par créer un tableau de valeurs booléennes en appliquant une condition à un tableau existant. Par exemple, pour créer un tableau booléen indiquant les éléments supérieurs à 5 dans un tableau `tableau` :

```
import numpy as np

# Création d'un tableau NumPy
tableau = np.array([1, 7, 3, 9, 4, 8])

# Création d'un tableau booléen en appliquant une condition
tableau_bool = tableau > 5  # Crée un tableau booléen avec True pour les valeurs > 5
```

Dans cet exemple, `tableau_bool` contiendra `[False, True, False, True, False, True]`.

6.5.2 Indexation à l'aide du Tableau Booléen

Une fois que vous avez créé un tableau booléen, vous pouvez l'utiliser pour extraire les éléments correspondants du tableau d'origine. Vous placez simplement le tableau booléen entre crochets pour effectuer la sélection. Les éléments associés à `True` seront extraits.

```
# Indexation à l'aide du tableau booléen
resultat = tableau[tableau_bool]  # Sélectionne les éléments > 5

# Affichage du résultat
print(resultat)  # Affichera [7, 9, 8]
```

L'indexation booléenne permet de filtrer rapidement et efficacement les données dans un tableau en fonction de conditions complexes. Elle est largement utilisée dans le traitement de données et l'analyse pour extraire des sous-ensembles spécifiques de données qui répondent à des critères spécifiques.

[16]:
```
A = np.array([[1, 2, 3], [4, 5, 6]])

print(A<5) # masque booléen

print(A[A < 5]) # sous-ensemble filtré par le masque booléen

A[A<5] = 4 # convertit les valeurs sélectionnées.
print(A)
```

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

6.6 Numpy : Mathématiques

6.6.1 Méthodes de bases (les plus utiles) de la classe ndarray

[34]:
```
A = np.array([[1, 2, 3], [4, 5, 6]])
print(A)
print("la somme des elements",A.sum()) # effectue la somme de tous les éléments␣
 ↪du tableau
print("la somme des elements colone",A.sum(axis=0)) # effectue la somme des␣
 ↪colonnes (somme sur éléments des les lignes)
print("la somme des elements ligne",A.sum(axis=1)) # effectue la somme des␣
 ↪lignes (somme sur les éléments des colonnes)
print("la somme cumulative",A.cumsum(axis=0)) # effectue la somme cumulée

print(A.prod()) # effectue le produit
print(A.cumprod()) # effectue le produit cumulé
```

```
print(A.min()) # trouve le minimum du tableau
print(A.max()) # trouve le maximum du tableau

print(A.mean()) # calcule la moyenne
print(A.std()) # calcule l'ecart type,
print(A.var()) # calcule la variance
```

```
[[1 2 3]
 [4 5 6]]
la somme des elements 21
la somme des elements colone [5 7 9]
la somme des elements ligne [ 6 15]
la somme cumulative [[1 2 3]
 [5 7 9]]
720
[  1   2   6  24 120 720]
1
6
3.5
1.707825127659933
2.9166666666666665
```

Une méthode tres importante : la méthode **argsort()**

[19]:
```
A = np.random.randint(0, 10, [5, 5]) # tableau aléatoire
print(A)
```

```
[[4 2 6 2 4]
 [7 7 4 0 1]
 [6 6 6 3 3]
 [2 0 7 7 3]
 [5 7 4 9 7]]
```

[20]:
```
print(A.argsort()) # retourne les index pour trier chaque ligne du tableau
```

```
[[1 3 0 4 2]
 [3 4 2 0 1]
 [3 4 0 1 2]
 [1 0 4 2 3]
 [2 0 1 4 3]]
```

[21]:
```
print(A[:,0].argsort()) # retourne les index pour trier la colonne 0 de A
```

```
[3 0 4 2 1]
```

[22]:
```
A = A[A[:,0].argsort(), :] # trie les colonnes du tableau selon la colonne 0.
A
```

[22]:
```
array([[2, 0, 7, 7, 3],
       [4, 2, 6, 2, 4],
       [5, 7, 4, 9, 7],
       [6, 6, 6, 3, 3],
       [7, 7, 4, 0, 1]])
```

6.6.2 Numpy Statistics

[ ]:
```
B = np.random.randn(3, 3) # nombres aléatoires 3x3

# retourne la matrice de corrélation de B
print(np.corrcoef(B))
```

```
[[ 1.         -0.63427277  0.99937797]
 [-0.63427277  1.         -0.66114251]
 [ 0.99937797 -0.66114251  1.        ]]
```

[24]:
```
# retourne la matrice de corrélation entre les lignes 0 et 1 de B
print(np.corrcoef(B[:,0], B[:, 1]))
```

```
[[1.         0.81847981]
 [0.81847981 1.        ]]
```

np.unique() :

[4]:
```
np.random.seed(0)
A = np.random.randint(0, 5, [5,5])
A
```

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

[16]:
```
np.unique(A)
```

[16]:
```
array([0, 1, 2, 3, 4])
```

[5]:
```
values, counts = np.unique(A, return_counts=True)
print(values, counts)
for i, j in zip(values[counts.argsort()], counts[counts.argsort()]):
    print(f'valeur {i} apparait {j}')
```

```
[0 1 2 3 4] [7 5 3 6 4]
valeur 2 apparait 3
valeur 4 apparait 4
valeur 1 apparait 5
```

```
valeur 3 apparait 6
valeur 0 apparait 7
```

Calculs statistiques en présence de données manquates (NaN)

[36]:
```
A = np.random.randn(5, 5)
A[0, 2] = np.nan # insere un NaN dans la matrice A

print('ratio NaN/zise:', (np.isnan(A).sum()/A.size)) # calcule la proportion de␣
↪NaN dans A

print('moyenne sans NaN:', np.nanmean(A)) # calcule la moyenne de A en ignorant␣
↪les NaN
```

```
ratio NaN/zise: 0.04
moyenne sans NaN: 0.1832816316588837
```

6.6.3 Algebre Linéaire

[38]:
```
A = np.ones((2,3))
B = np.ones((3,3))

print(A.T) # transposé de la matrice A (c'est un attribut de ndarray)
```

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

[39]:
```
print(A.dot(B)) # produit matriciel A.B
```

```
[[3. 3. 3.]
 [3. 3. 3.]]
```

[45]:
```
A = np.random.randint(0, 10, [3, 3])

print('det=', np.linalg.det(A)) # calcule le determinant de A
print('inv A:\n', np.linalg.inv(A)) # calcul l'inverse de A
```

```
det= 61.00000000000001
inv A:
 [[ 0.08196721  0.63934426 -0.60655738]
 [-0.13114754 -0.62295082  0.7704918 ]
 [ 0.21311475  0.26229508 -0.37704918]]
```

[44]:
```
val, vec = np.linalg.eig(A)
print('valeur propre:\n', val) # valeur propre
print('vecteur propre:\n', vec) # vecteur propre
```

```
valeur propre:
 [ 8.91371956 -0.86320273  1.94948316]
vecteur propre:
 [[-0.27183844 -0.6838339   0.39494311]
 [-0.4097407  -0.15279739 -0.73291062]
 [-0.87075623  0.71345929  0.55395123]]
```

6.6.4 Fonctions mathématiques

[13]:
```
A = np.linspace(1,10,10)
print(np.exp(A)) # calclule l'exponontielle pour tous les element d'un tableau
print(np.log(A)) # calclule le logaritheme pour tous les element d'un tableau
print(np.cos(A)) # calclule le cosinus pour tous les element d'un tableau
print(np.sin(A)) # calclule le sinus pour tous les element d'un tableau
```

```
[2.71828183e+00 7.38905610e+00 2.00855369e+01 5.45981500e+01
 1.48413159e+02 4.03428793e+02 1.09663316e+03 2.98095799e+03
 8.10308393e+03 2.20264658e+04]
[0.         0.69314718 1.09861229 1.38629436 1.60943791 1.79175947
 1.94591015 2.07944154 2.19722458 2.30258509]
[ 0.54030231 -0.41614684 -0.9899925  -0.65364362  0.28366219  0.96017029
  0.75390225 -0.14550003 -0.91113026 -0.83907153]
[ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155
  0.6569866   0.98935825  0.41211849 -0.54402111]
```

6.7 Exercices

1. Array Creation and Manipulation: Create a NumPy array of shape (3, 4) filled with random integers between 1 and 10. Then, reshape it into a (2, 6) array.
2. Array Concatenation: Create two NumPy arrays of the same shape (3, 3) filled with random numbers. Concatenate them vertically using vstack() and horizontally using hstack().
3. Indexing and Slicing: Create a NumPy array of shape (5, 5) filled with sequential integers from 1 to 25. Use array indexing and slicing to extract the middle 3x3 subarray.
4. Array Operations: Create a NumPy array of shape (4, 4) filled with random integers. Calculate the sum, mean, and standard deviation of the array's elements.
5. Boolean Indexing: Create a NumPy array of random integers between 1 and 100. Use boolean indexing to extract all values greater than 50.
6. Sur l'image ci dessous, effectuer un slicing pour ne garder que la moitié de l'image (en son centre) et remplacer tous les pixels > 150 par des

[ ]:
```
pixels = 255
from scipy import misc
import matplotlib.pyplot as plt
face = misc.face(gray=True)
plt.imshow(face, cmap=plt.cm.gray)
```

```
plt.show()
face.shape
```

7. Standardisez la matrice suivante, c'est a dire effectuez le calcul suivant : $A = \frac{A - mean(A_{colonne})}{std(A_{colonne})}$

[ ]:
```
np.random.seed(0)
A = np.random.randint(0, 100, [10, 5])
A
```

7 TP 7: Scipy

Scipy contient des modules tres puissants pour le machine learning, l'anayse de données, les time series, etc. Ce notebook vous montre quelques unes des fonctions les plus utiles

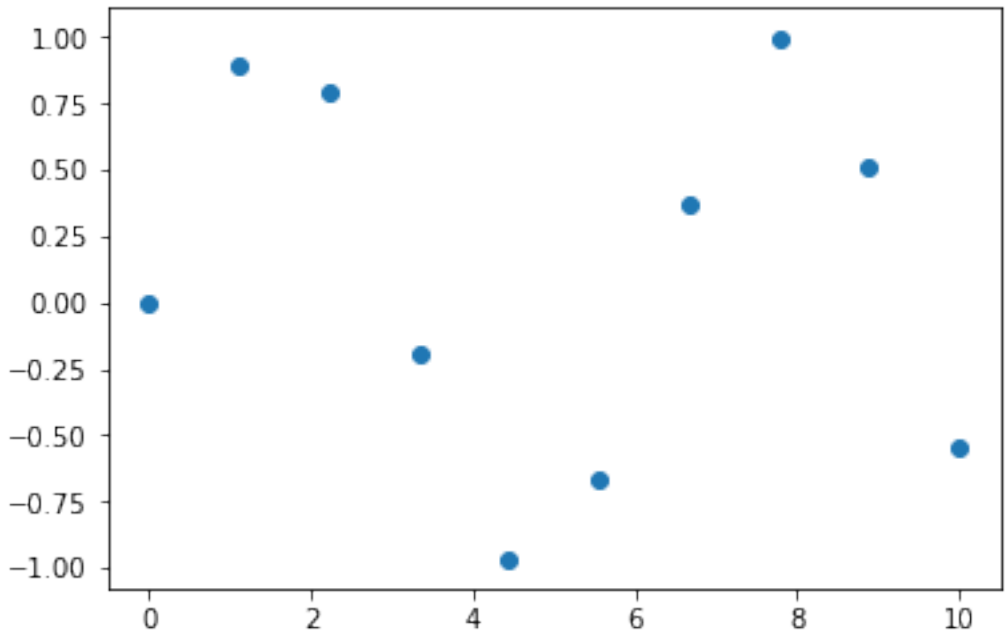
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

7.1 Interpolation

Interpoler un signal est parfois tres utile s'il vous manque des données dans un Dataset. Mais c'est une technique dangereuse, qui peut parfois transformer la réalité des choses ! Dans cette partie, nous allons interpoler les données de cet échantillon de données.

[2]:
```
# Création d'un Dataset
x = np.linspace(0, 10, 10)
y = np.sin(x)
plt.scatter(x, y)
```

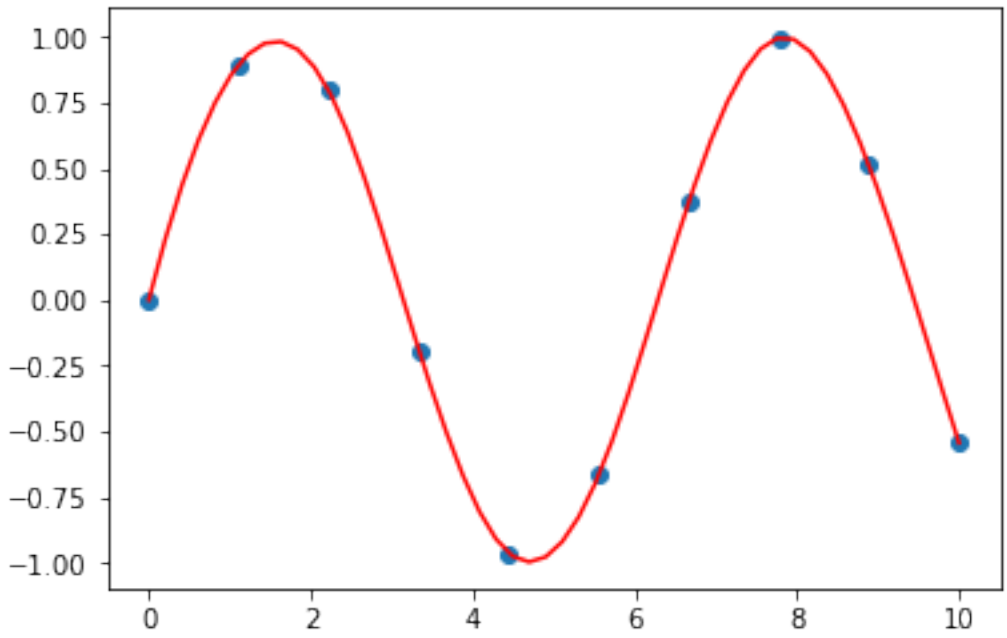
[2]:
```
<matplotlib.collections.PathCollection at 0x1da399b0ac8>
```



[3]:
```
from scipy.interpolate import interp1d
```

[4]:
```
# création de la fonction interpolation f
f = interp1d(x, y, kind='cubic')

# résultats de la fonction interpolation f sur de nouvelles données
new_x = np.linspace(0, 10, 50)
result = f(new_x)

# visualisation avec matplotlib
plt.scatter(x, y)
plt.plot(new_x, result, c='r')
```

[4]:
```
[<matplotlib.lines.Line2D at 0x1da399de128>]
```



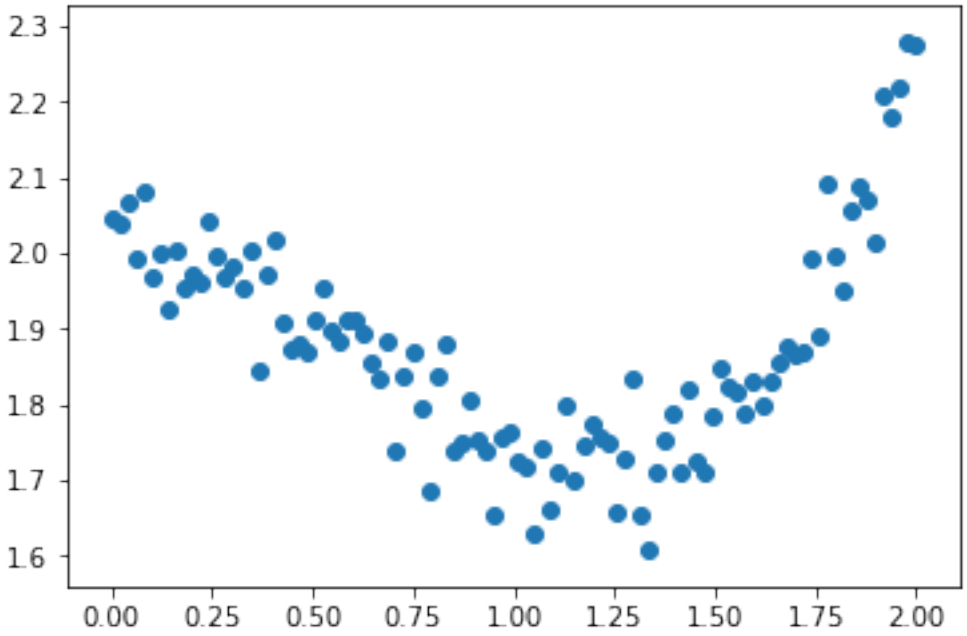
7.2 Optimisation

On trouve beaucoup de fonctions dans le module **optimize**. Certaines permettent de faire des minimisations locales, ou globales, d'autres permette de développer des modeles statistiques avec la méthode des moindres carrés. On trouve également des fonctions pour faire de la programmation linéaire.

7.2.1 curve_fit

[5]:
```
# Création d'un Dataset avec du bruit "normal"
x = np.linspace(0, 2, 100)
y = 1/3*x**3 - 3/5 * x**2 + 2 + np.random.randn(x.shape[0])/20
plt.scatter(x, y)
```

[5]:
```
<matplotlib.collections.PathCollection at 0x1da3a0bd940>
```



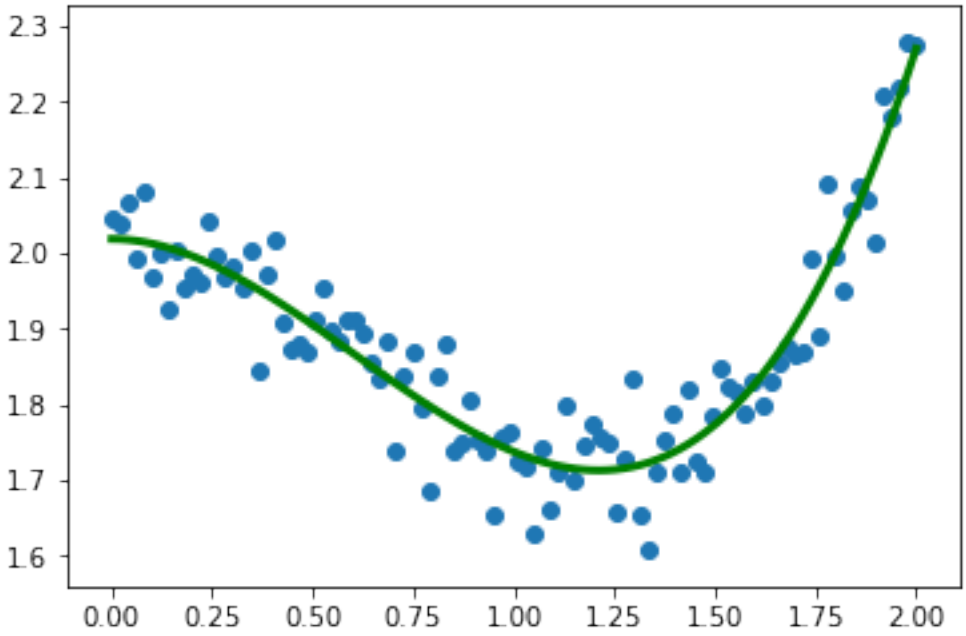
[6]:
```
# Définition d'un modele statistique sensé "coller" au dataset ci-dessus
def f (x, a, b, c, d):
    return a * x**3 + b * x**2 + c * x + d
```

[7]:
```
from scipy import optimize
```

[8]:
```
# curve_fit permet de trouver les parametres du modele f grace a la méthode des␣
 ↪moindres carrés
params, param_cov = optimize.curve_fit(f, x, y)
```

[9]:
```
# Visualisation des résultats.
plt.scatter(x, y)
plt.plot(x, f(x, params[0], params[1], params[2], params[3]), c='g', lw=3)
```

[9]:
```
[<matplotlib.lines.Line2D at 0x1da3a37c160>]
```



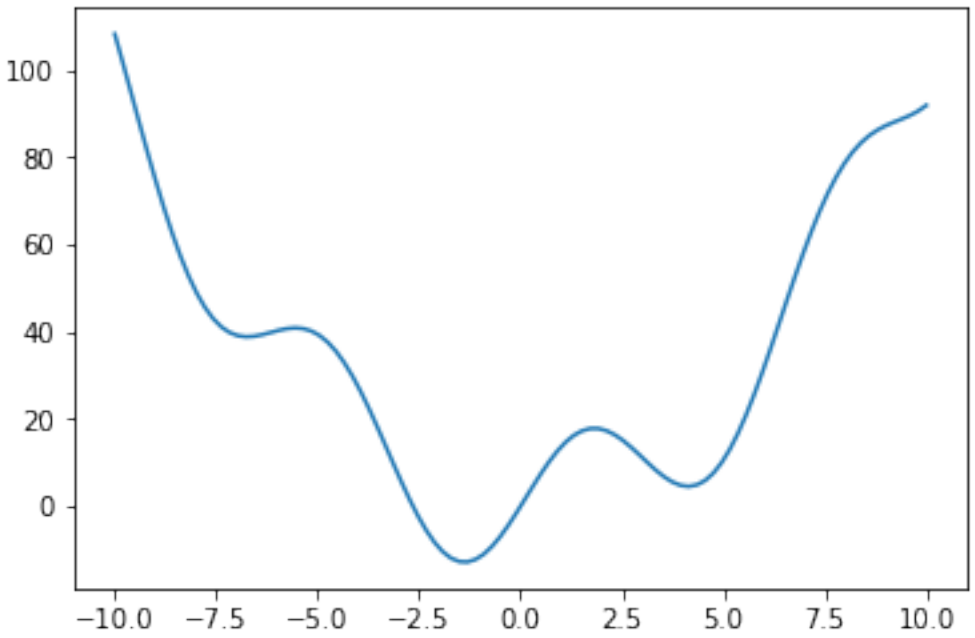
7.2.2 Minimisation 1D

la fonction **optimize.minimize** est utile pour trouver un minimum local dans une fonction a N dimensions

[10]:
```
# Définition d'une fonction a 1 Dimension
def f (x):
    return x**2 + 15*np.sin(x)
```

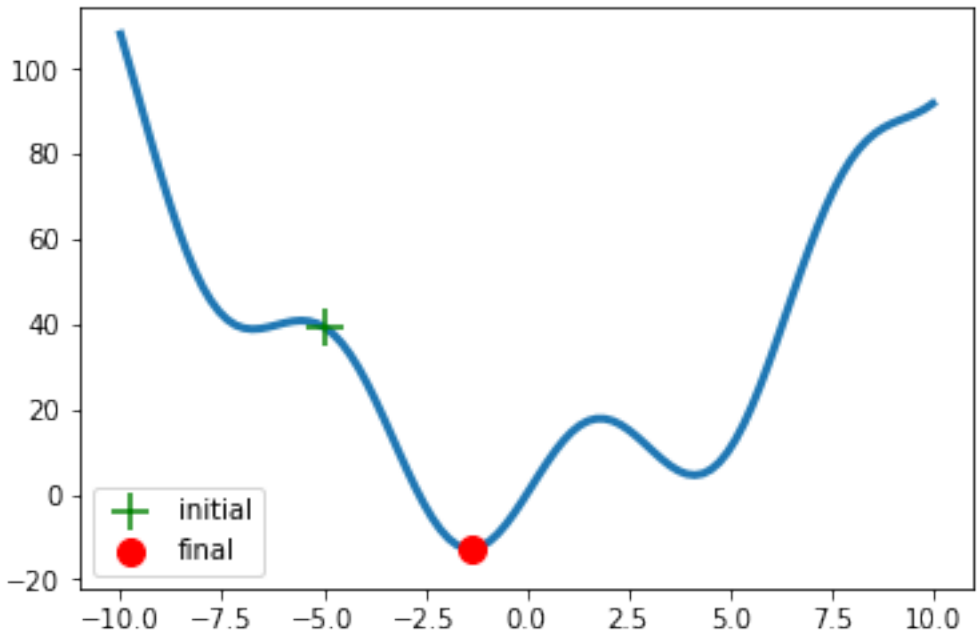
[11]:
```
# Visualisation de la fonction
x = np.linspace(-10, 10, 100)
plt.plot(x, f(x))
```

[11]:
```
[<matplotlib.lines.Line2D at 0x1da3a81ec88>]
```



[12]:
```
# Définition d'un point x0 pour l'algorithme de minimisation
x0=-5
result = optimize.minimize(f, x0=x0).x # résultat de la minimisation
```

[13]:
```
# Visualisation du résultat

plt.plot(x, f(x), lw=3, zorder=-1) # Courbe de la fonction
plt.scatter(x0, f(x0), s=200, marker='+', c='g', zorder=1, label='initial') #␣
 ↪point initial
plt.scatter(result, f(result), s=100, c='r', zorder=1, label='final') # point␣
 ↪final
plt.legend()
plt.show()
```

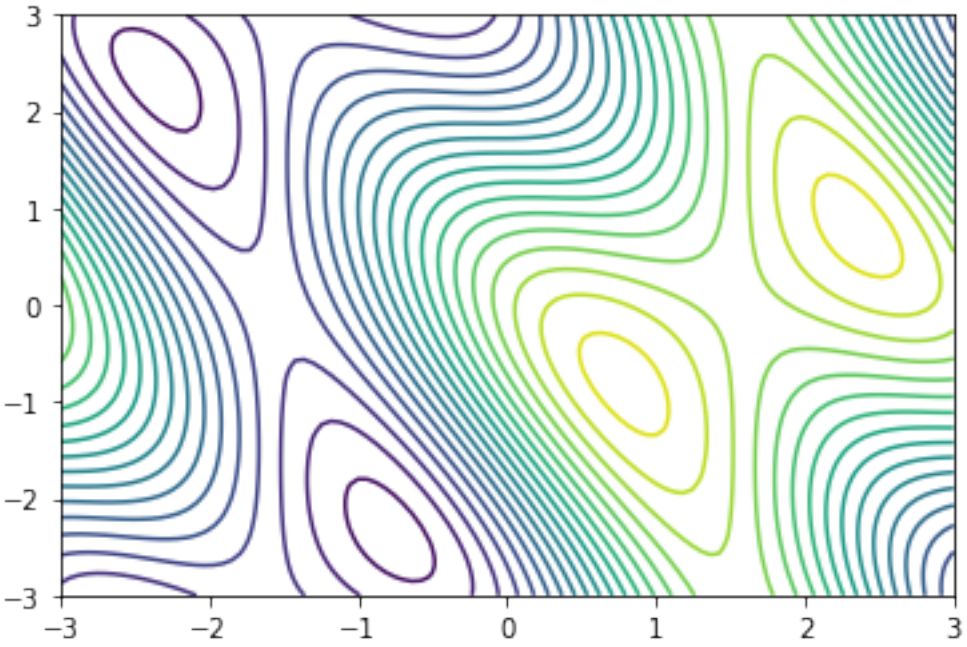


7.2.3 Minimisation 2D

[14]:
```
# Définition d'une fonction 2D. X est un tableau numpy a 2-Dimension
def f (x):
    return np.sin(x[0]) + np.cos(x[0]+x[1])*np.cos(x[0])
```

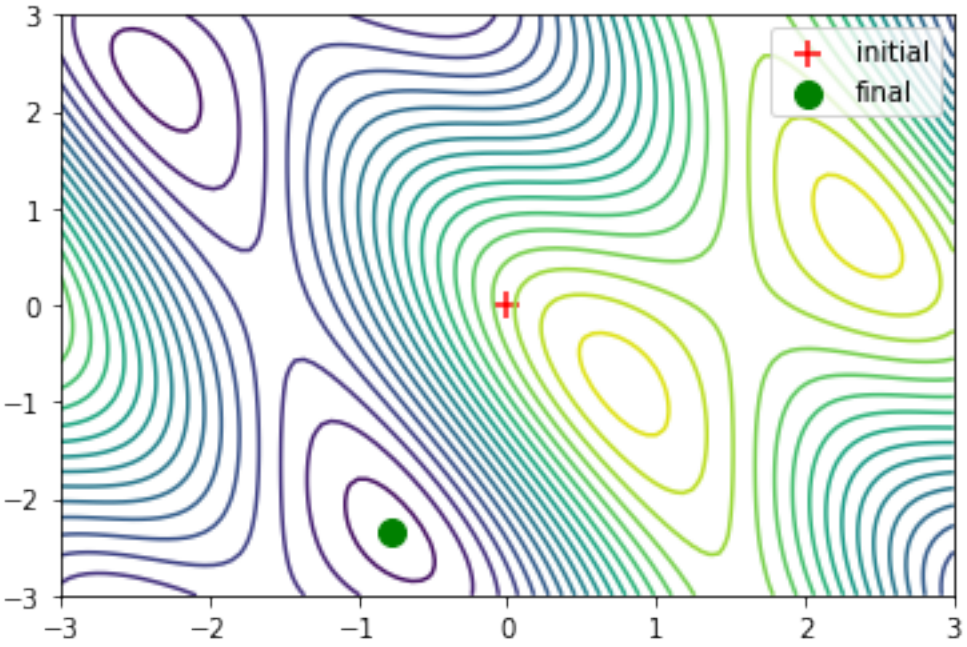
[15]:
```
# Génération de la fonction sur un espace 2D.
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
x, y = np.meshgrid(x, y)

# Visualisation de la fonction
plt.contour(x, y, f(np.array([x, y])), 20)
```

[15]:
```
<matplotlib.contour.QuadContourSet at 0x1da3a907630>
```



[16]:
```
# Placement d'un point x0 initial aux coordonées (0,0)
x0 = np.zeros((2, 1))

# Minimisation de la fonction
result = optimize.minimize(f, x0=x0).x
print('le minimum est aux coordonées', result) # imprimer le résultat

# Visualisation du résultat
plt.contour(x, y, f(np.array([x, y])), 20) # fonction 2D
plt.scatter(x0[0], x0[1], marker='+', c='r', s=100, label='initial') # Point de␣
 ↪départ
plt.scatter(result[0], result[1], c='g', s=100, label='final') # Point final
plt.legend()
plt.show()
```

```
le minimum est aux coordonées [-0.78539917 -2.35619341]
```



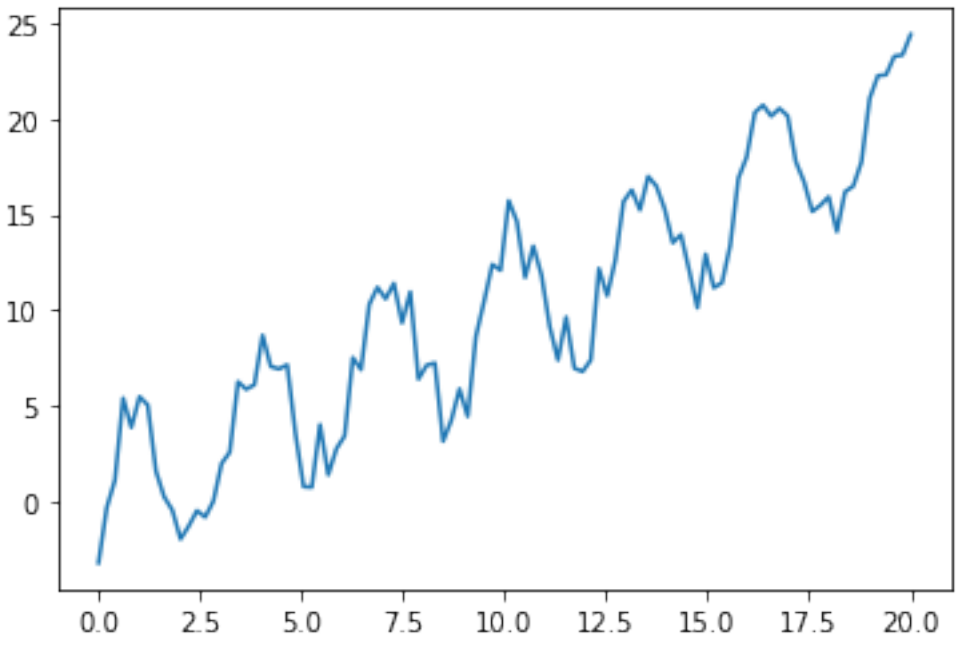
7.3 Traitement du signal

Le module **scipy.signal** contient beaucoup de fonctions de convolution et de filtres pour faire du traitement du signal. La fonction **signal.detrend** est parfaite pour éliminer une tendance linéaire dans un signal. Utile pour beaucoup d'applications !

Le module **scipy.fftpack** contient des fonctions tres puissantes et simples d'utilisation pour effectuer des transformations de Fourier

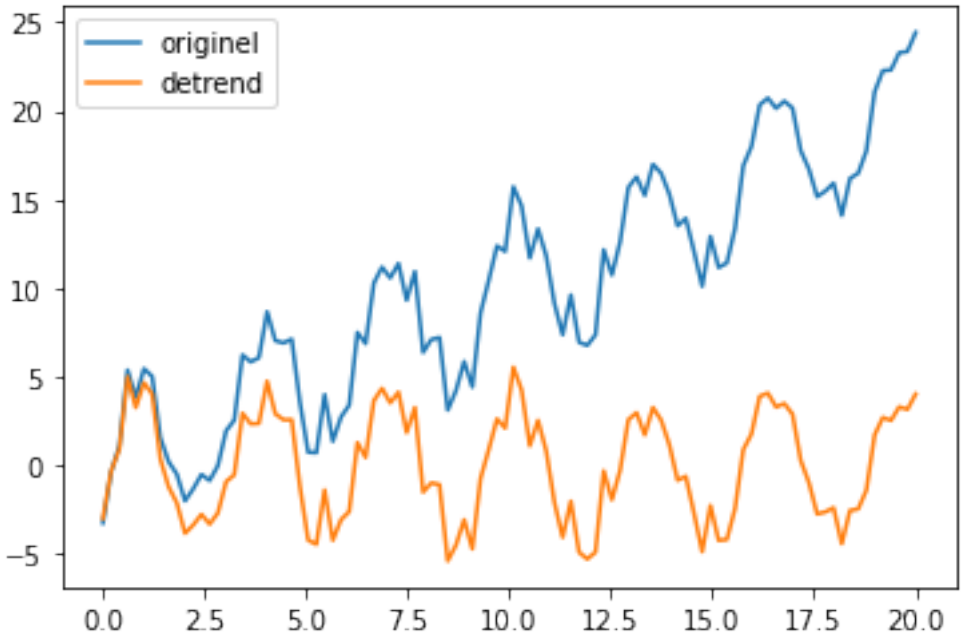
[3]:
```
# Création d'un Dataset avec une tendance linéaire
x = np.linspace(0, 20, 100)
y = x + 8*np.sin(x)*np.cos(x) +np.random.randn(x.shape[0])
plt.plot(x, y)
```

[3]:
```
[<matplotlib.lines.Line2D at 0x7f98d93bfd90>]
```



[4]:
```
from scipy import signal
```

[5]:
```
# Élimination de la tendance linéaire
new_y = signal.detrend(y)

# Visualisation des résultats
plt.plot(x, y, label='originel')
plt.plot(x, new_y, label='detrend')
plt.legend()
plt.show()
```



7.4 Transformation de Fourier (FFT)

La transformation de Fourier est une technique mathématique puissante et normalement complexe a mettre en oeuvre. Heureusement **scipy.fftpack** rend cette technique tres simple a implémenter

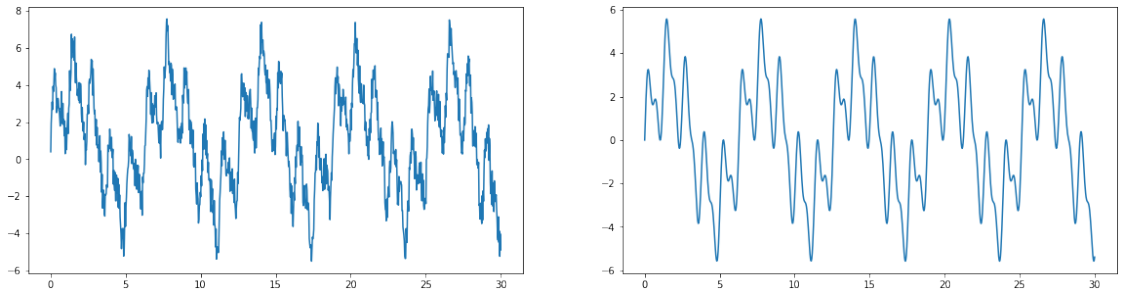
La transformation de Fourier permet d'analyser les **fréquences** qui composent un signal **périodique** (qui se répete avec le temps). Cette opération produit un graphique que l'on appelle **Spectre**.

Une fois le **Spectre** généré, il est possible de filtrer les bruits indésirables, ou bien de sélectionner seulement certaines fréquences, ou d'en atténuer d'autres... les possibilités sont infinies.

Dans l'exemple ci-dessous, nous voyons comment filtrer un signal noyé dans du bruit.

[20]:
```
# Création d'un signal périodique noyé dans du bruit.
x = np.linspace(0, 30, 1000)
y_clean = 3*np.sin(x) + 2*np.sin(5*x) + np.sin(10*x)
y = y_clean+np.random.random(x.shape[0])*2

plt.subplots(1,2,figsize=(20,5))
plt.subplot(1,2,1)
plt.plot(x, y)
plt.subplot(1,2,2)
plt.plot(x, y_clean)
```

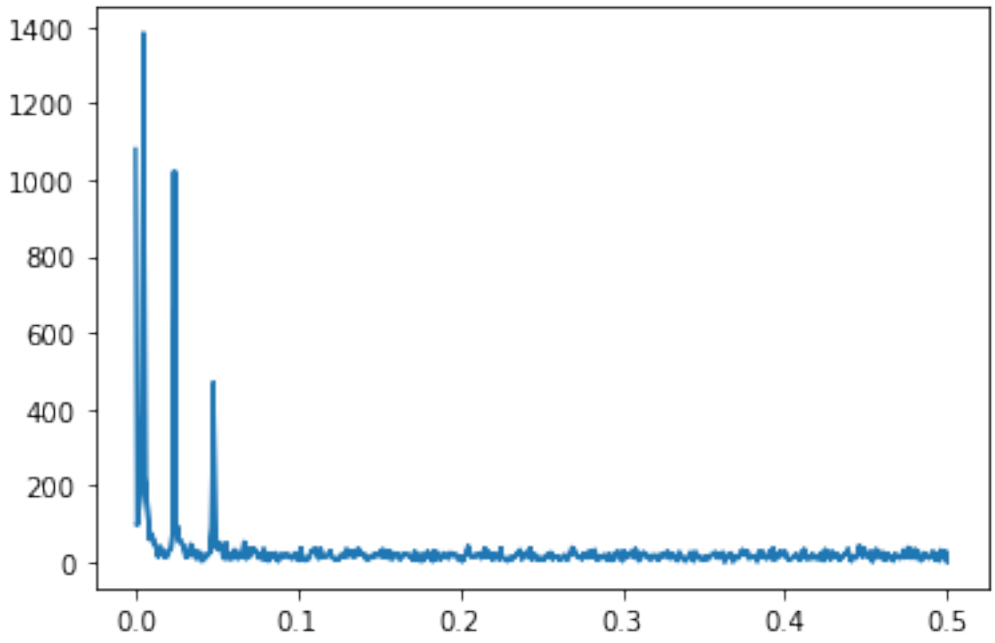
[20]:
```
[<matplotlib.lines.Line2D at 0x7fc78b20d5b0>]
```



[11]:
```
from scipy import fftpack
```

[21]:
```
# création des variables Fourier et Fréquences, qui permettent de construire le␣
 ↪spectre du signal.
fourier = fftpack.fft(y)
power = np.abs(fourier) # la variable power est créée pour éiminer les␣
 ↪amplitudes négatives
frequences = fftpack.fftfreq(y.size)
plt.plot(np.abs(frequences), power)
```

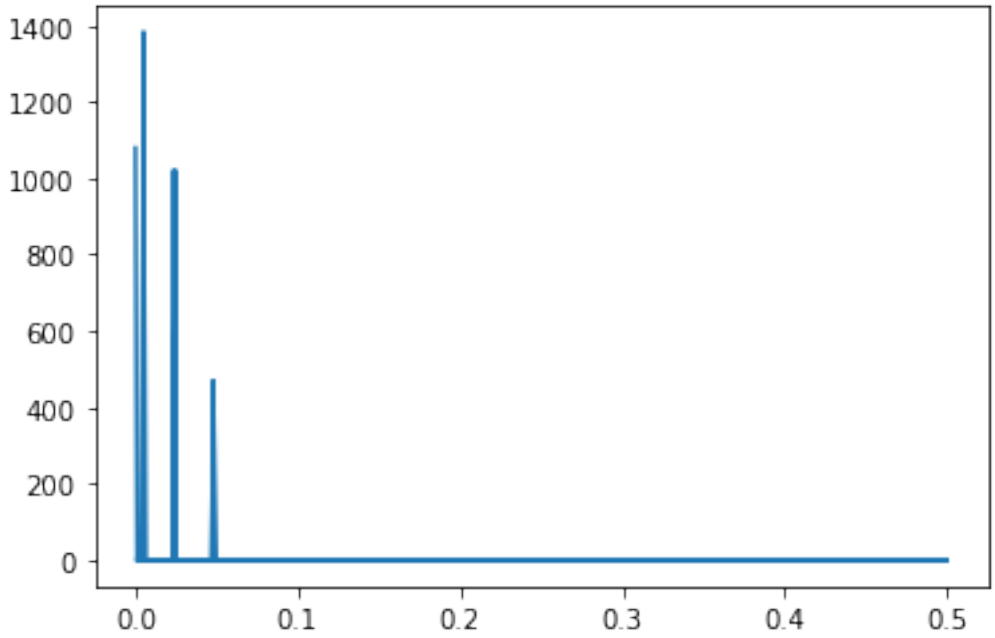
[21]:
```
[<matplotlib.lines.Line2D at 0x7fc78b22faf0>]
```



[22]:
```
# filtre du spectre avec du boolean indexing de Numpy
fourier[power<400] = 0
```

```
# Visualisation du spetre propre
plt.plot(np.abs(frequences), np.abs(fourier))
```

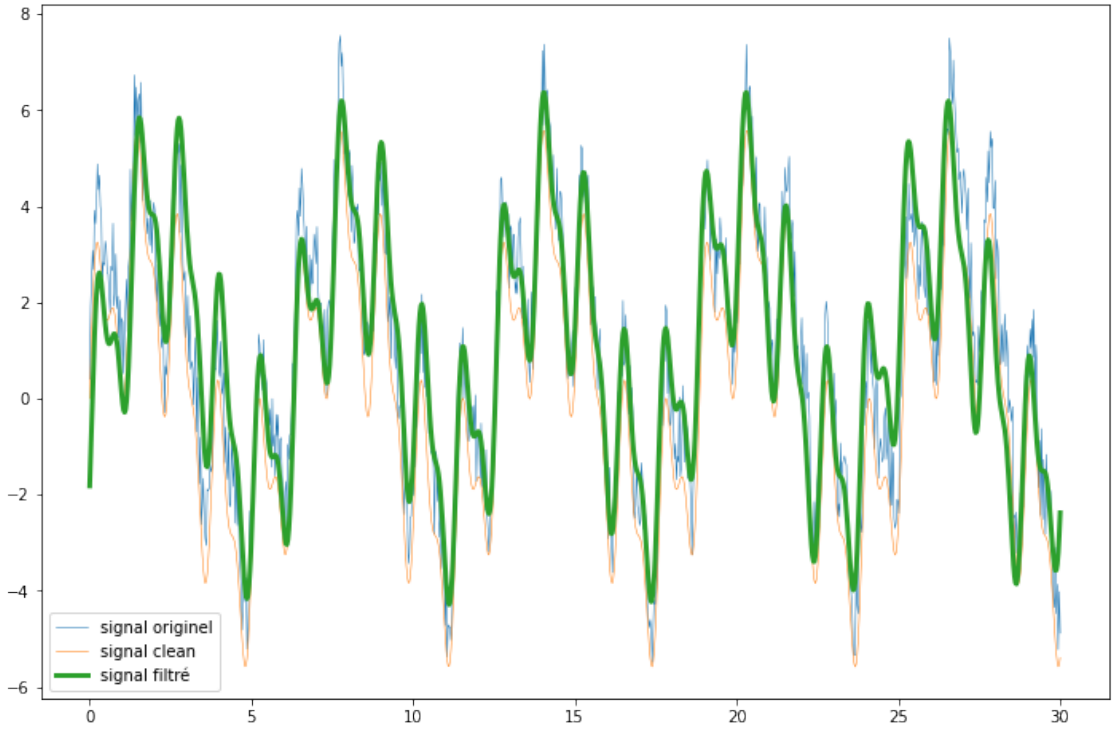
[22]: [<matplotlib.lines.Line2D at 0x7fc78b459af0>]



[23]:
```
# Transformation de Fourier Inverse: genere un nouveau signal temporel depuis le␣
↪spectre filtré
filtered_signal = fftpack.ifft(fourier)
```

[24]:
```
# Visualisation des résultats

plt.figure(figsize=(12, 8))
plt.plot(x, y, lw=0.5, label='signal originel')
plt.plot(x, y_clean, lw=0.5, label='signal clean')
plt.plot(x, filtered_signal, lw=3, label='signal filtré')
plt.legend()
plt.show()
```

```
/Users/mac/opt/anaconda3/lib/python3.8/site-
packages/matplotlib/cbook/__init__.py:1289: ComplexWarning: Casting complex
values to real discards the imaginary part
  return np.asarray(x, float)
```



7.5 Traitement d'image

scipy.ndimage propose de nombreuses actions pour le traitement d'images: convolutions, filtres de Gauss, méthode de mesures, et morphologie.

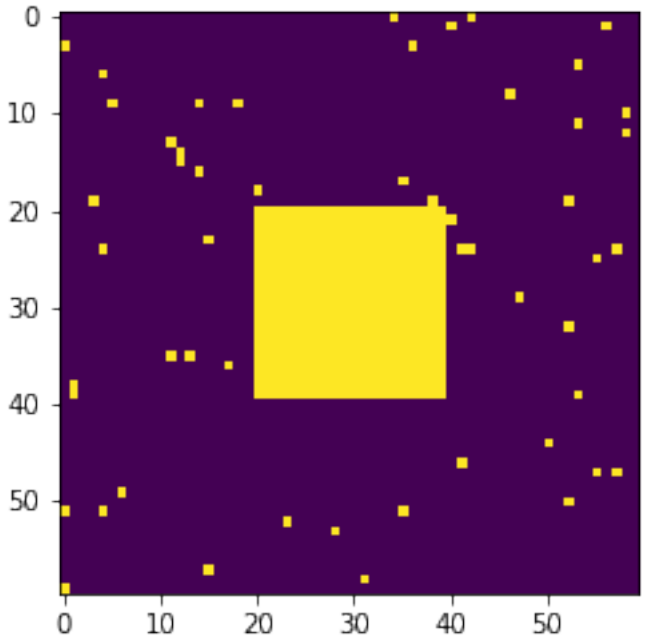
La morphologie est une technique qui permet de transformer une matrice (et donc une image) par le déplacement d'une structure sur chaque pixel de l'image. Lorsqu'un pixel "blanc" est visité, la structure peut effectuer une opération: - de dilation: imprime des pixels - d'érosion : efface des pixels

Cette technique peut-etre utile pour nettoyer une image des artefacts qui peuvent la composer.

[25]:
```
from scipy import ndimage
```

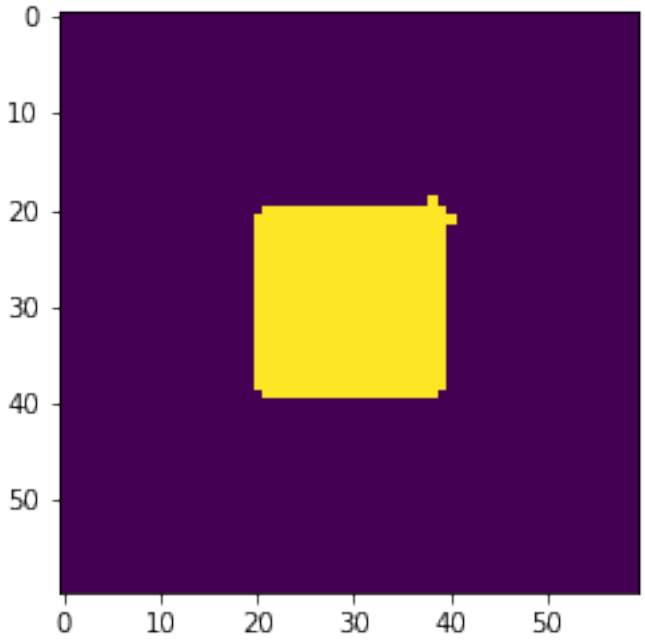
[26]:
```
# Création d'une image avec quelques artefacts
np.random.seed(0)
X = np.zeros((60, 60))
X[20:-20, 20:-20] = 1
X[np.random.randint(0,60,60),np.random.randint(0,60,60)] = 1 #ajout d'artefacts␣
↪aléatoires
plt.imshow(X)
```

[26]:
```
<matplotlib.image.AxesImage at 0x7fc78b79eb50>
```



[27]:
```
# opération de binary_opening = érosion puis dilation
open_X = ndimage.binary_opening(X)
plt.imshow(open_X)
```

[27]: <matplotlib.image.AxesImage at 0x7fc78b9c6910>

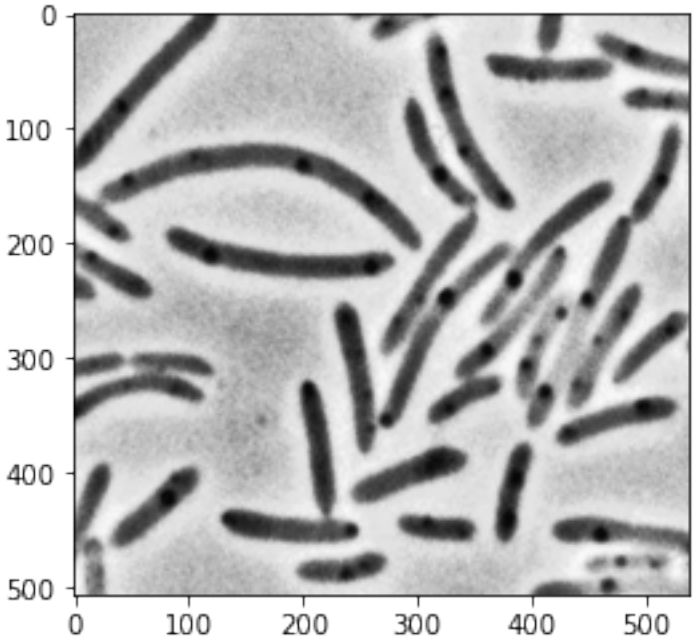


7.6 Application (cas réel)

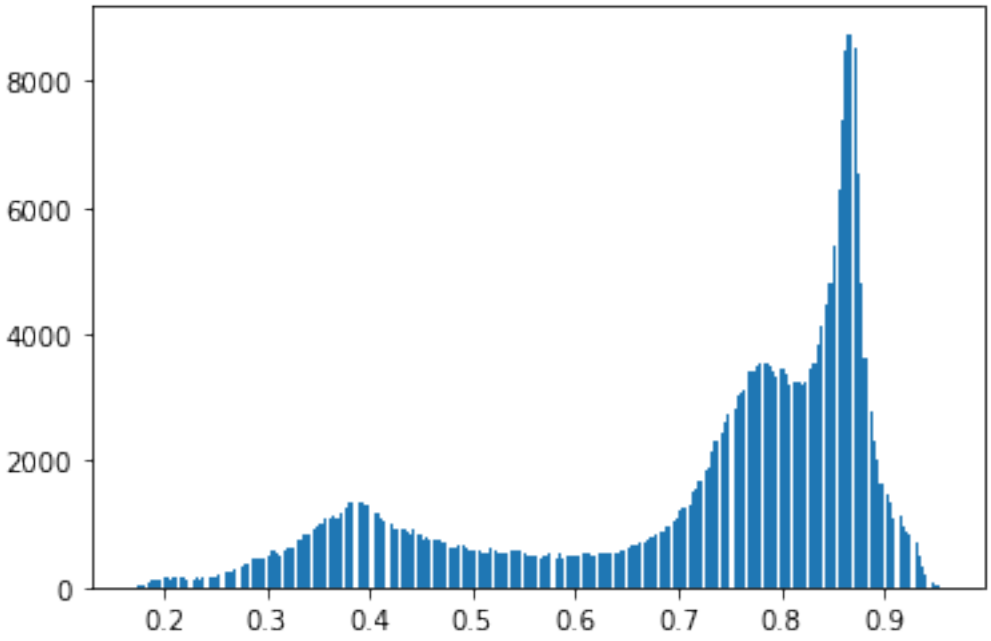
Vous trouvez cette image dans le pack de la formation:

[28]:
```
# importer l'image avec pyplot
image = plt.imread('Data/bacteria.png')
image = image[:,:,0] # réduire l'image en 2D
plt.imshow(image, cmap='gray') # afficher l'image
image.shape
```

[28]:
```
(507, 537)
```

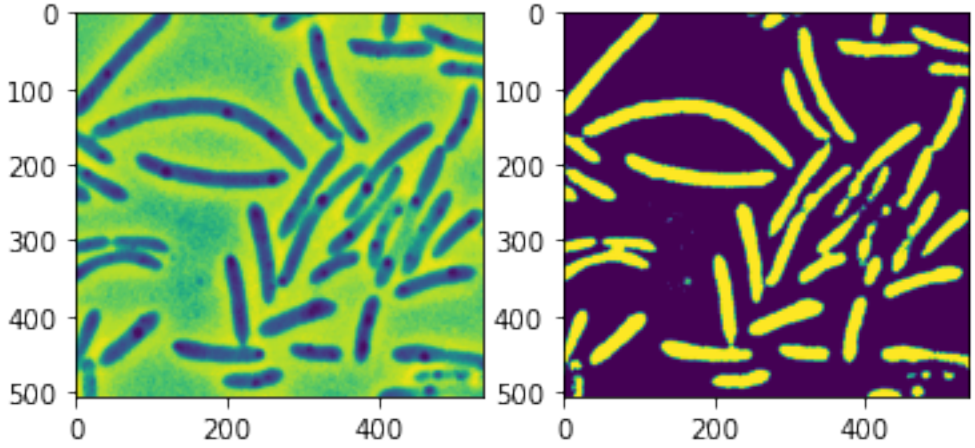


[29]:
```
# copy de l'image, puis création d'un histogramme
image_2 = np.copy(image)
plt.hist(image_2.ravel(), bins=255)
plt.show()
```



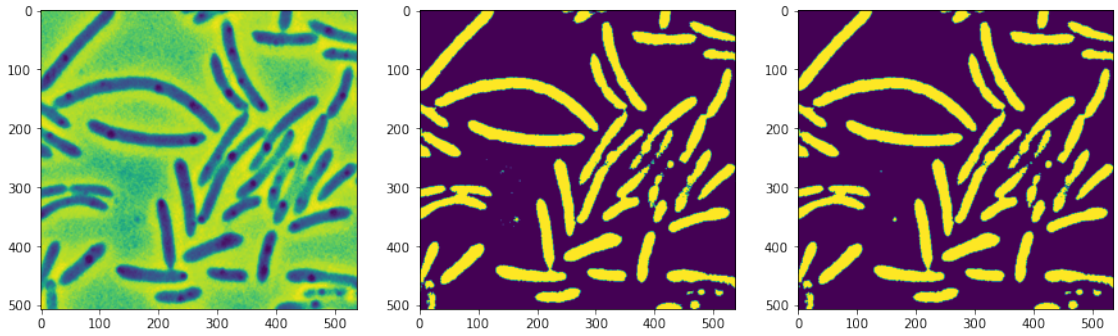
[36]:
```
# boolean indexing: création d'une image binaire
image_clean= image<0.6
plt.subplot(1,2,1)
plt.imshow(image)
plt.subplot(1,2,2)
plt.imshow(image_clean)
```

[36]: <matplotlib.image.AxesImage at 0x7fc7889c6fa0>



[37]:
```
# morphologie utilisée pour enlever les artefacts
image_clean2 = ndimage.binary_opening(image_clean)
plt.figure(figsize=(15, 5))
plt.subplot(1,3,1)
plt.imshow(image)
plt.subplot(1,3,2)
plt.imshow(image_clean)
plt.subplot(1,3,3)
plt.imshow(image_clean2)
```

[37]:
```
<matplotlib.image.AxesImage at 0x7fc78a4888b0>
```

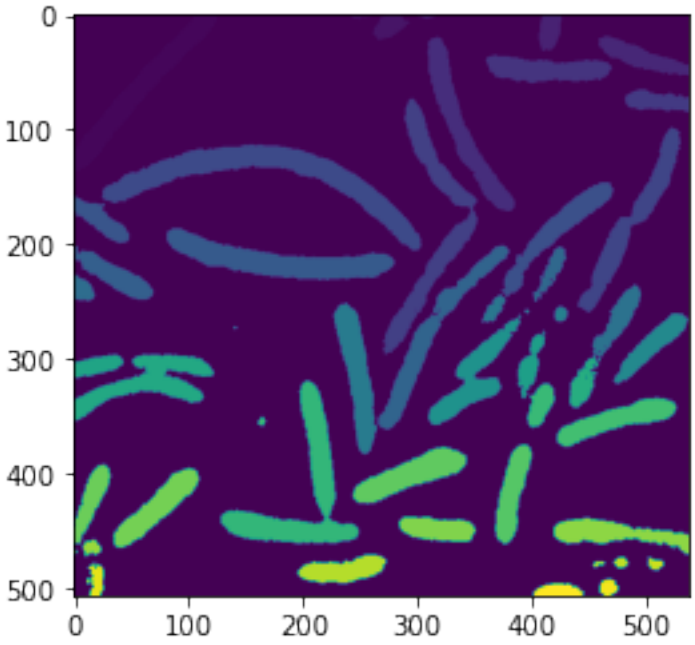


[38]:
```
# Segmentation de l'image: label_image contient les différents labels et␣
↪n_labels est le nombre de labels
label_image, n_labels = ndimage.label(image_clean2)
print(f'il y a {n_labels} groupes')
```

```
il y a 53 groupes
```

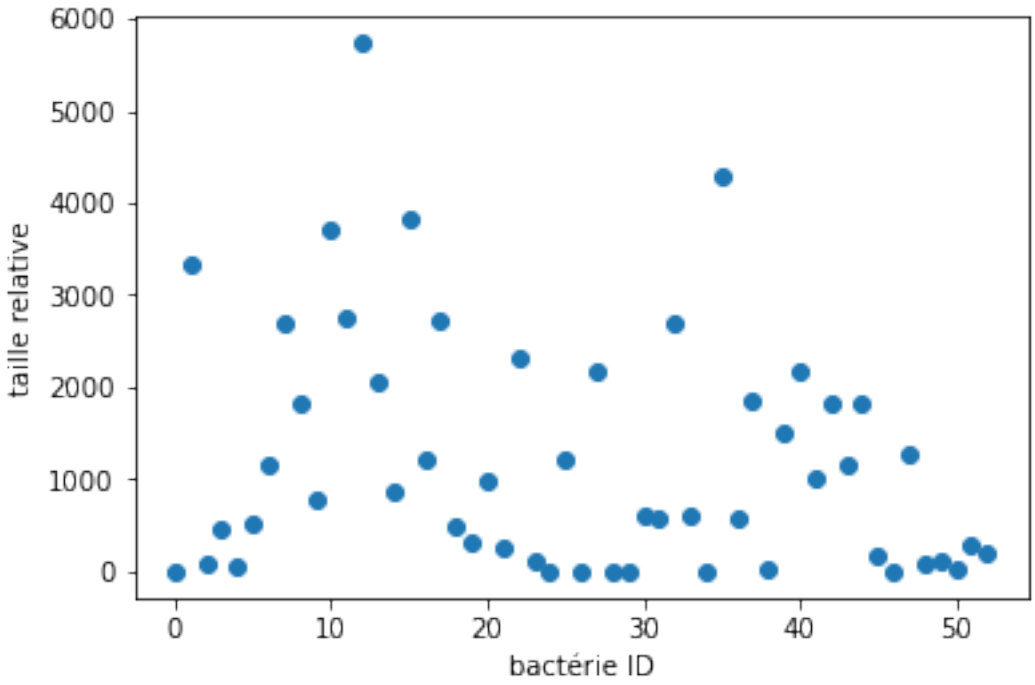
[33]:
```
# Visualisation de l'image étiquetée
plt.imshow(label_image)
```

[33]:
```
<matplotlib.image.AxesImage at 0x7fc78b7f5340>
```



[39]:
```
# Mesure de la taille de chaque groupes de label_images (fait la somme des␣
↪pixels)
sizes = ndimage.sum(image_clean2, label_image, range(n_labels))
```

[40]:
```
# Visualisation des résultats
plt.scatter(range(n_labels), sizes)
plt.xlabel('bactérie ID')
plt.ylabel('taille relative')
plt.show()
```



[ ]:

8 TP 8: Matplotlib

Bien sûr ! Voici une explication du cycle de vie pour la création de figures en Python à l'aide de la bibliothèque Matplotlib en français académique :

8.1 Cycle de Vie pour la Création de Figures avec Matplotlib

Lorsque vous créez des figures à l'aide de la bibliothèque Matplotlib en Python, il est important de suivre un cycle de vie organisé pour obtenir des graphiques propres et bien présentés. Ce cycle de vie comprend les étapes suivantes :

8.1.1 Création de la Figure avec `plt.figure()` et `figsize`

La première étape consiste à créer une nouvelle figure en utilisant la fonction `plt.figure()` de Matplotlib. Cette fonction permet de définir la taille de la figure à l'aide du paramètre `figsize`, qui spécifie les dimensions de la figure en pouces (largeur, hauteur).

```
import matplotlib.pyplot as plt

# Création d'une nouvelle figure avec une taille personnalisée
plt.figure(figsize=(8, 6))
```

La création de la figure est la première étape cruciale pour préparer l'espace où vous allez créer votre graphique.

8.1.2 Création du Graphique avec `plt.plot()`

Une fois la figure créée, vous pouvez ajouter des graphiques à l'intérieur en utilisant la fonction `plt.plot()`. Cette fonction permet de tracer des courbes, des lignes ou des points en fonction de vos données.

```
# Création d'un graphique à l'intérieur de la figure
plt.plot(x, y, label='Courbe de données')
```

Ici, `x` et `y` représentent les données que vous souhaitez tracer, et `label` est utilisé pour donner un nom à la courbe, ce qui sera utile pour ajouter des légendes plus tard.

8.1.3 Ajout d'Extras (Titres, Axes, Légendes)

Pour rendre votre figure plus informative, vous pouvez ajouter des extras tels que des titres, des étiquettes d'axes et des légendes. Ces éléments aident à expliquer le contenu de votre graphique.

```
# Ajout d'un titre au graphique
plt.title('Graphique de Données')

# Ajout d'étiquettes aux axes x et y
plt.xlabel('Axe X')
plt.ylabel('Axe Y')

# Ajout d'une légende pour la courbe
plt.legend()
```

8.1.4 Affichage de la Figure avec `plt.show()`

Une fois que vous avez configuré votre figure avec les graphiques et les extras nécessaires, vous pouvez afficher la figure à l'aide de la fonction `plt.show()`. Cette fonction affiche la figure à l'écran.

```
# Affichage de la figure
plt.show()
```

C'est la dernière étape du cycle de vie. Après cela, votre graphique sera affiché à l'écran dans une fenêtre séparée ou dans votre environnement de développement.

8.2 Pyplot

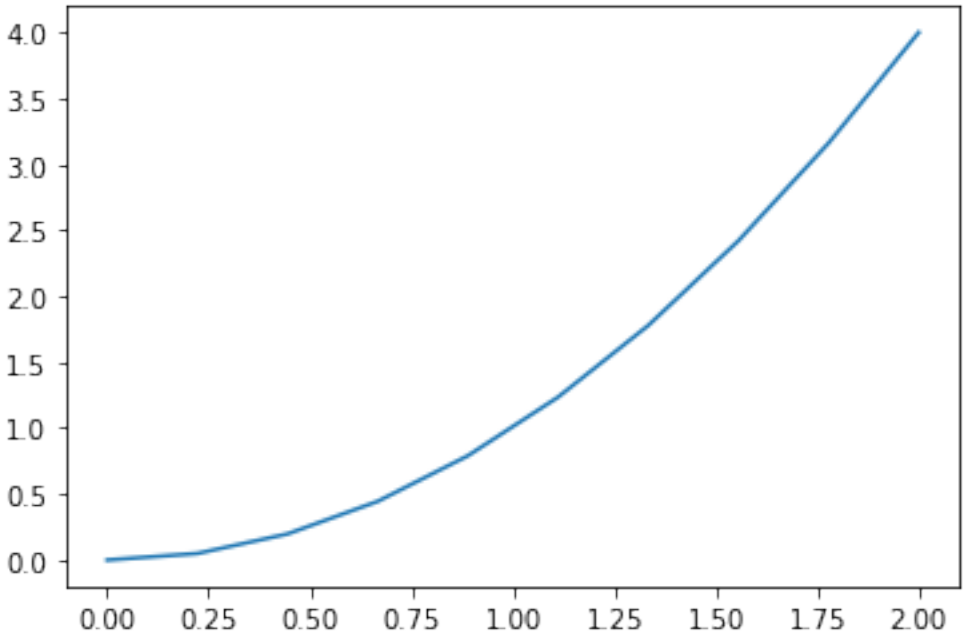
pyplot est principalement destiné aux tracés interactifs et aux cas simples de génération de tracés programmatiques :

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

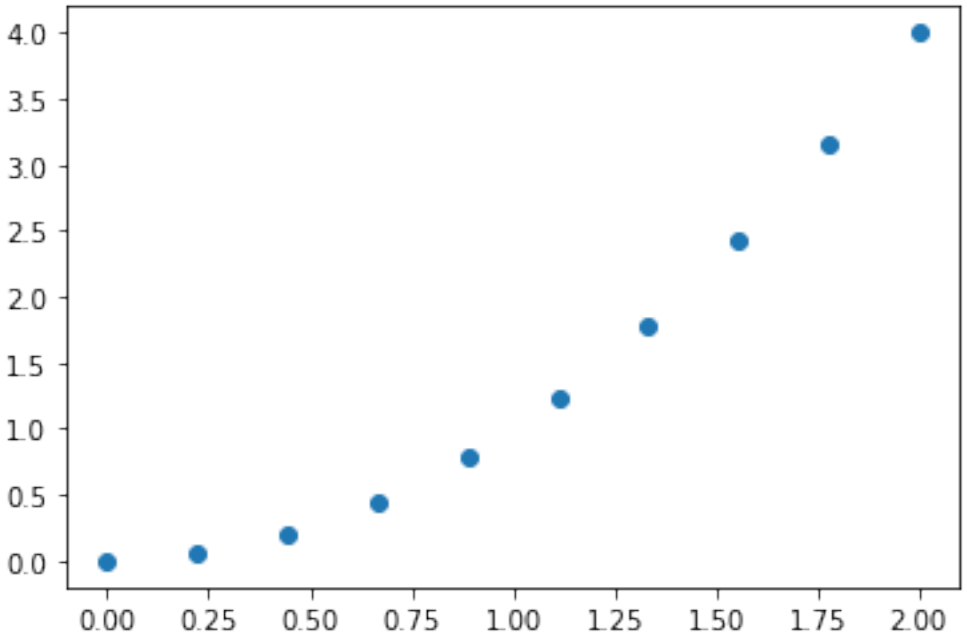
8.2.1 Graphiques simples

[5]:
```
X = np.linspace(0, 2, 10)
y = X**2
```

```
plt.plot(X, y)
plt.show()
```



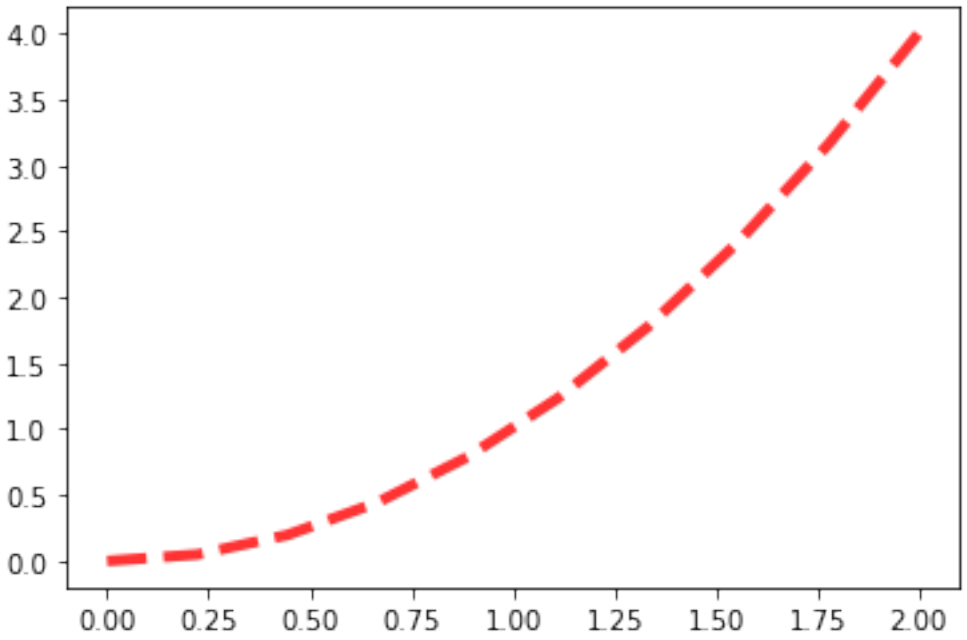
[6]:
```
plt.scatter(X, y)
plt.show()
```



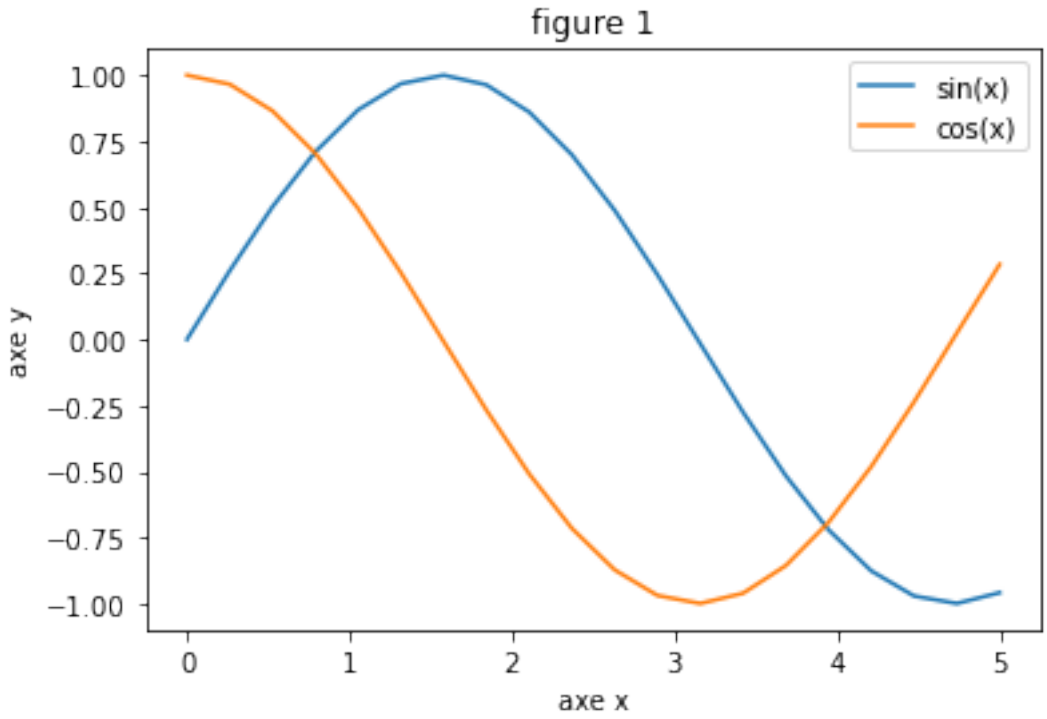
8.2.2 Styles Graphiques

Il existe beaucoup de styles a ajouter aux graphiques. Voici les plus importants a retenir : - **c** : couleur de la ligne - **lw** : epaisseur de la ligne (pour les graphiques plot) - **ls** : style de la ligne (pour les graphiques plot) - **size** : taille du point (pour les graphiques scatter) - **marker** : style de points (pour les graphiques scatter) - **alpha** : transparence du graphique

[9]:
```
plt.plot(X, y, c='red', lw=4, ls='--', alpha=0.8)
plt.show()
```



[14]:
```
X = np.linspace(0, 5, 20)

plt.figure() # Création d'une figure
plt.plot(X, np.sin(X), label='sin(x)') # premiere courbe
plt.plot(X, np.cos(X), label='cos(x)') # deuxieme courbe
# Extra information
plt.title('figure 1') # titre
plt.xlabel('axe x') # axes
plt.ylabel('axe y') # axes
plt.legend() # legend

plt.savefig('figure.png') # sauvegarde la figure dans le repertoire de travail
plt.show() # affiche la figure
```

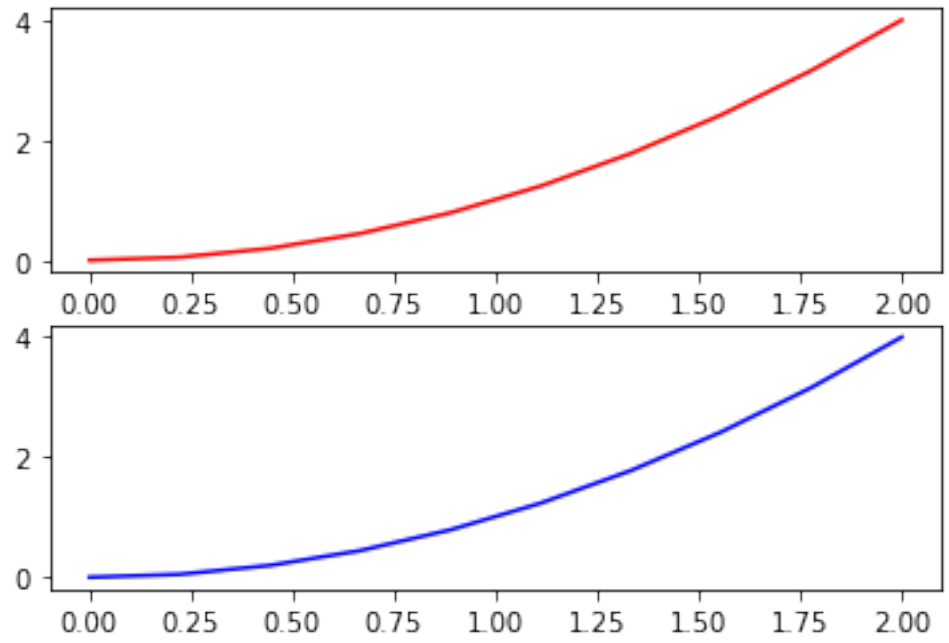


8.2.3 Subplots

Les subplots sont un autre éléments a ajouter pour créer plusieurs graphiques sur une meme figure

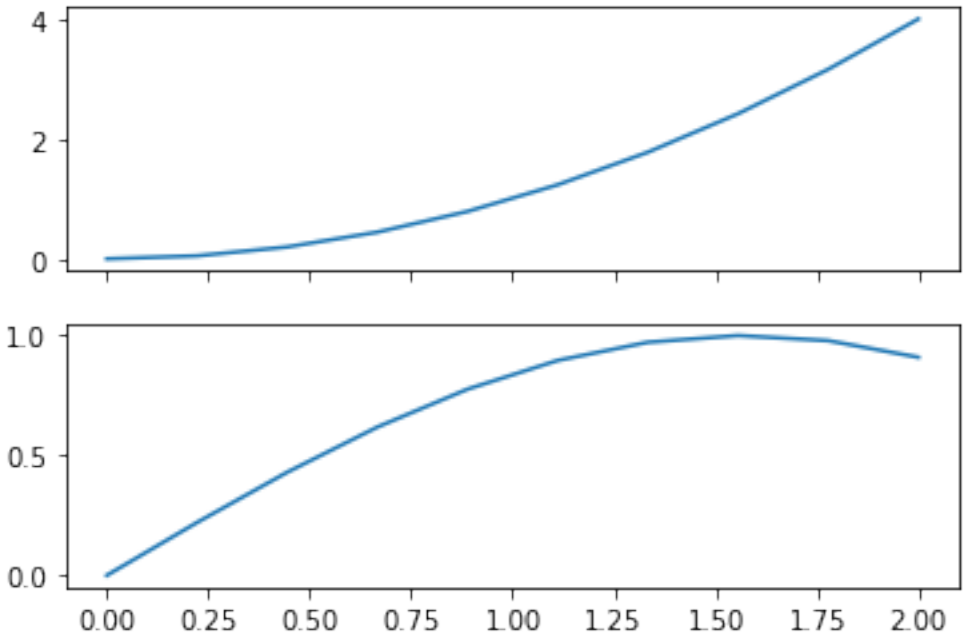
[13]:
```
plt.subplot(2, 1, 1)
plt.plot(x, y, c='red')
plt.subplot(2, 1, 2)
plt.plot(x, y, c='blue')
```

[13]:
```
[<matplotlib.lines.Line2D at 0x7f4610e2b940>]
```



8.2.4 Méthode orientée objet

[15]:
```
fig, ax = plt.subplots(2, 1, sharex=True) # partage le meme axe pour les subplots
ax[0].plot(x, y)
ax[1].plot(x, np.sin(x))
plt.show()
```



Exercice 2 Créez une fonction "graphique" qui permet de tracer sur une seule et meme figure une série de graphiques issue d'un dictionnaire contenant plusieurs datasets :

[19]:
```
def graphique(dataset):
  # Votre code ici...
  return

# Voici le dataset utilisé
dataset = {f"experience{i}": np.random.randn(100) for i in range(4)}
```

[20]:
```
# SOLUTION
def graphique(data):


#<--------Votre code ici --------->

    plt.show()
```

[21]:
```
graphique(dataset)
```

8.3 Matplotlib Graphiques importants

[4]:
```
import numpy as np
import matplotlib.pyplot as plt
```

8.3.1 Graphique de Classification (Scatter())

Les diagrammes de dispersion sont utilisés pour observer les relations entre les variables et utilisent des points pour représenter la relation entre elles. La méthode scatter() de la bibliothèque matplotlib est utilisée pour dessiner un nuage de points. Les diagrammes de dispersion sont largement utilisés pour décrire la relation entre les variables et comment le changement de l'une affecte l'autre.

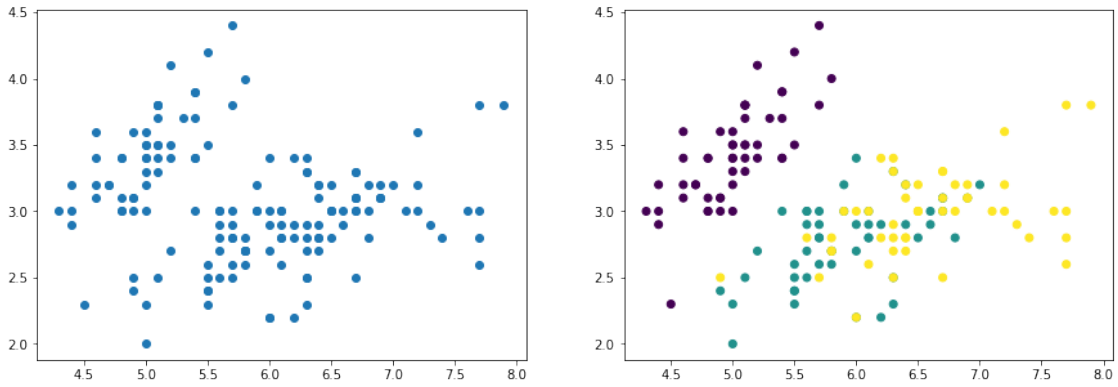
[5]:
```
from sklearn.datasets import load_iris
```

[24]:
```
iris = load_iris()
x = iris.data
y = iris.target

print(f'x contient {x.shape[0]} exmples et {x.shape[1]} variables')
print(f'il y a {np.unique(y).size} classes')
```

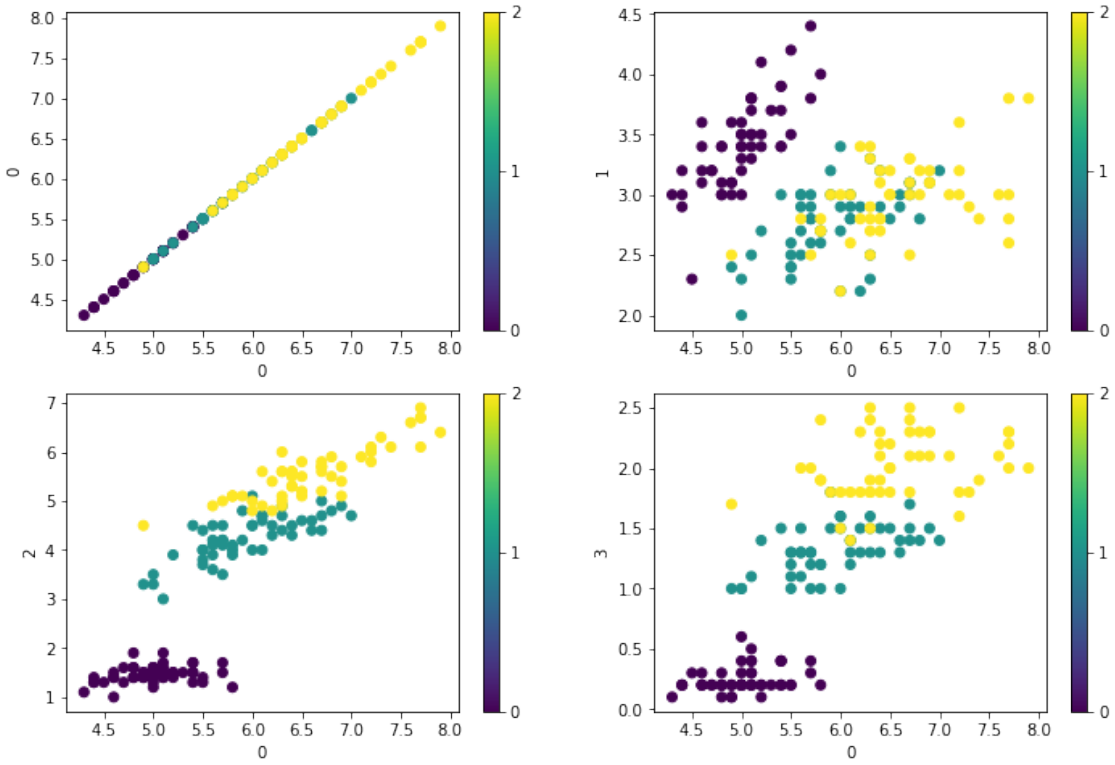
```
x contient 150 exmples et 4 variables
il y a 3 classes
```

[17]:
```
plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.scatter(x[:, 0], x[:, 1])
plt.subplot(1, 2, 2)
plt.scatter(x[:, 0], x[:, 1],c=y)
```

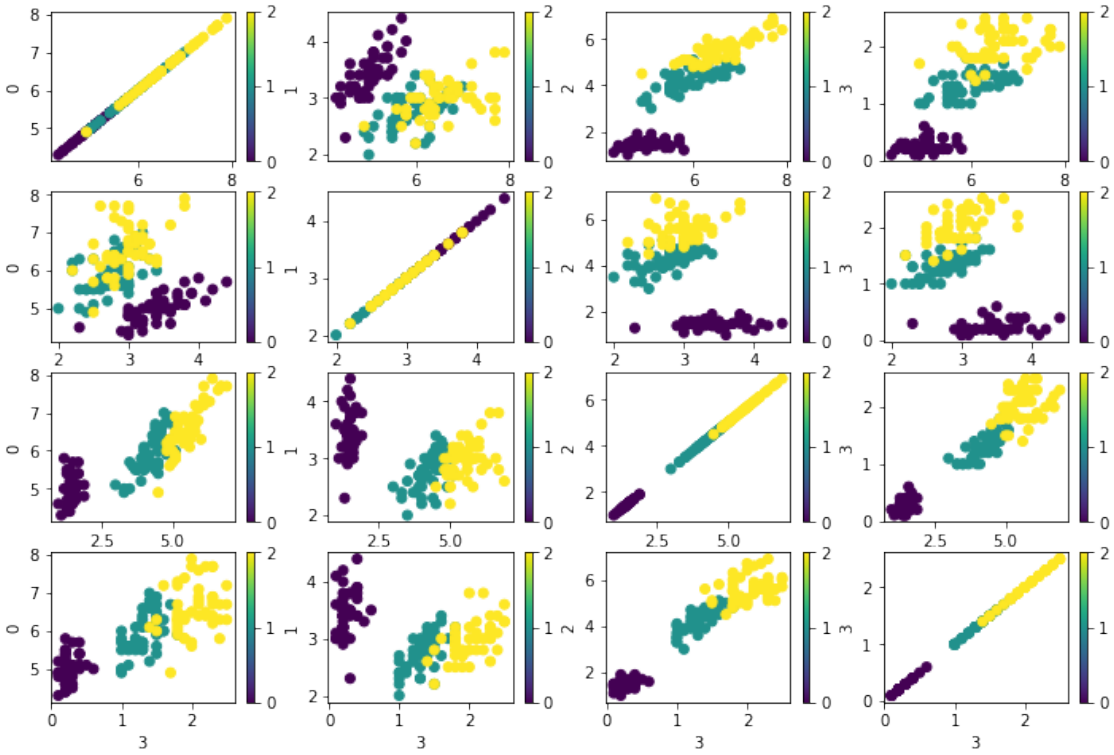
[17]:
```
<matplotlib.collections.PathCollection at 0x7fbde1309220>
```



[10]:
```
n = x.shape[1]
plt.figure(figsize=(12, 8))
for i in range(n):
    plt.subplot(n//2, n//2, i+1)
    plt.scatter(x[:, 0], x[:, i], c=y)
    plt.xlabel('0')
    plt.ylabel(i)
    plt.colorbar(ticks=list(np.unique(y)))
plt.show()
```



[11]:
```
n = x.shape[1]
plt.figure(figsize=(12, 8))
for j in range(n):
    for i in range(n):
        plt.subplot(n, n, (n*j)+i+1)
        plt.scatter(x[:, j], x[:, i], c=y)
        plt.xlabel(j)
        plt.ylabel(i)
        plt.colorbar(ticks=list(np.unique(y)))
plt.show()
```

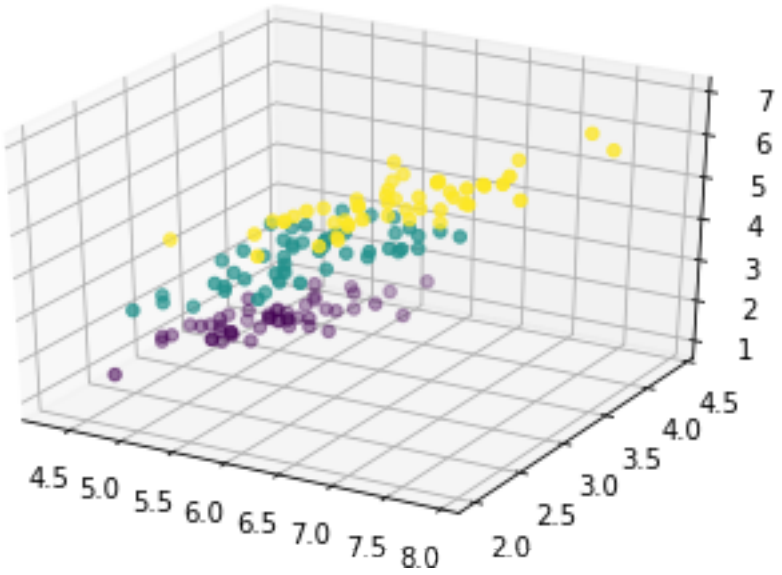


8.3.2 Graphiques 3D

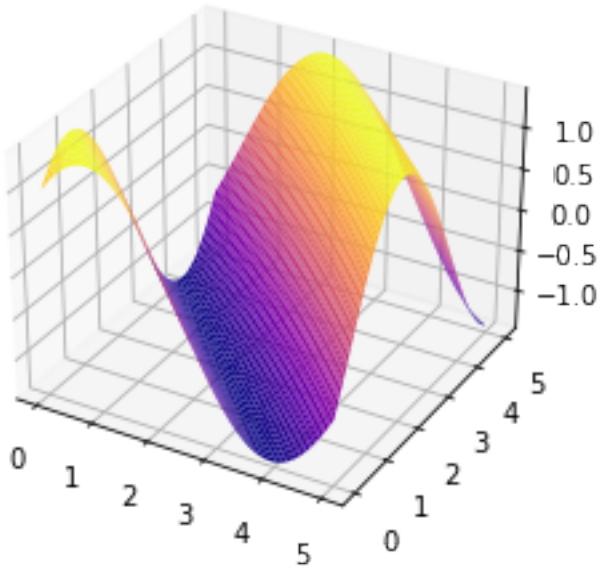
[ ]:
```
from mpl_toolkits.mplot3d import Axes3D
```

[7]:
```
ax = plt.axes(projection='3d')
ax.scatter(x[:, 0], x[:, 1], x[:,2], c=y)
```

[7]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7ff395f80358>
```



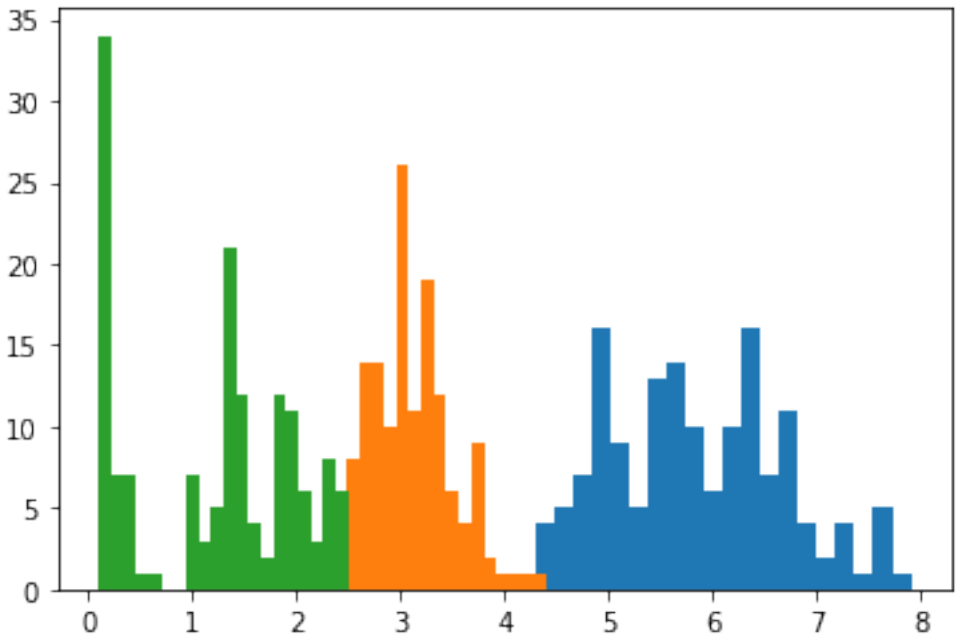
[18]:
```
f = lambda x, y: np.sin(x+y) + np.cos(x+y)

X = np.linspace(0, 5, 50)
Y = np.linspace(0, 5, 50)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)

ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, cmap='plasma')
plt.show()
```



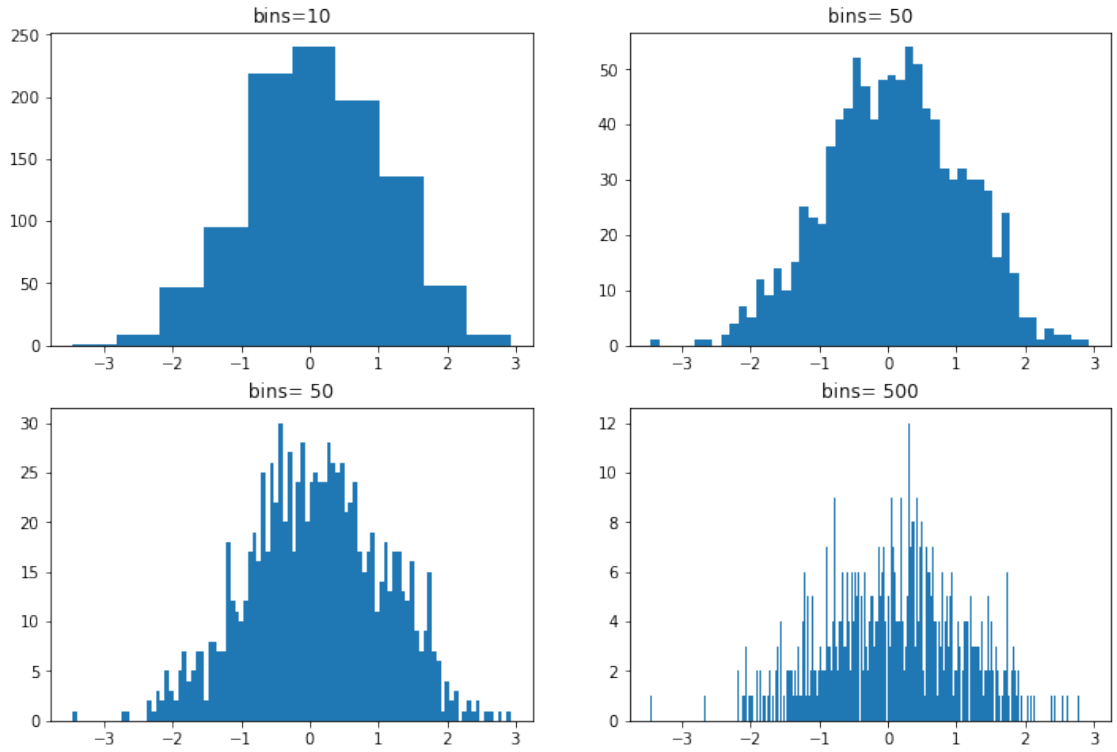
8.3.3 Histogrammes

[29]:
```
plt.figure()
plt.hist(x[:,0],bins=20)
plt.hist(x[:,1],bins=20)
plt.hist(x[:,3],bins=20)
```

[29]:
```
(array([34.,  7.,  7.,  1.,  1.,  0.,  0.,  7.,  3.,  5., 21., 12.,  4.,
         2., 12., 11.,  6.,  3.,  8.,  6.]),
 array([0.1 , 0.22, 0.34, 0.46, 0.58, 0.7 , 0.82, 0.94, 1.06, 1.18, 1.3 ,
        1.42, 1.54, 1.66, 1.78, 1.9 , 2.02, 2.14, 2.26, 2.38, 2.5 ]),
 <BarContainer object of 20 artists>)
```

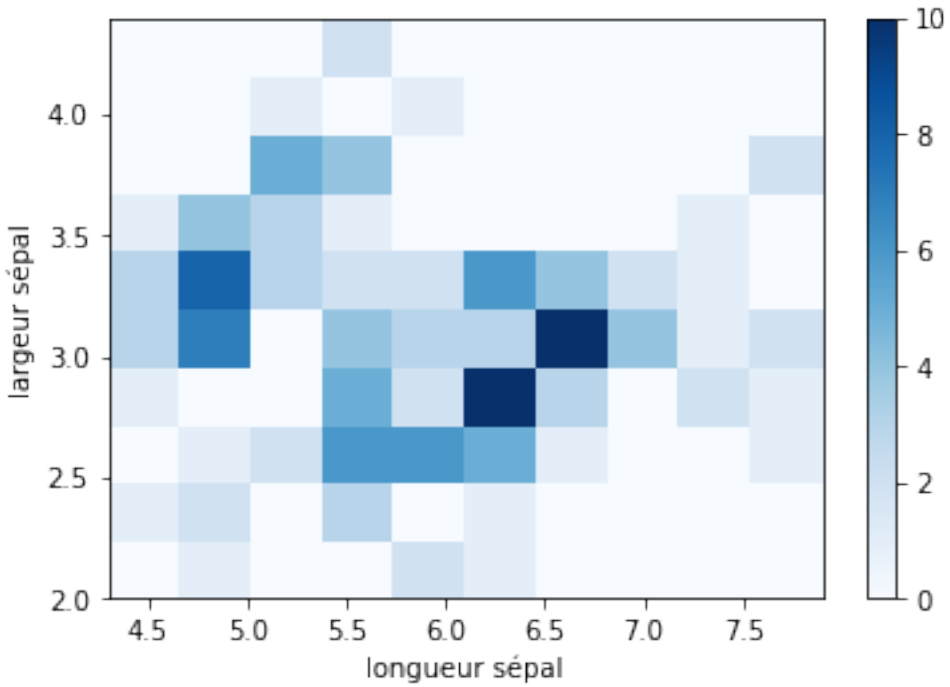


[21]:
```
x = np.random.randn(1000)

plt.figure(figsize=(12, 8))
plt.subplot(221)
plt.hist(x, bins=10)
plt.title('bins=10')
plt.subplot(222)
plt.hist(x, bins=50)
plt.title('bins= 50')
plt.subplot(223)
plt.hist(x, bins=100)
plt.title('bins= 50')
plt.subplot(224)
plt.hist(x, bins=500)
plt.title('bins= 500')
plt.show()
```



[12]:
```
x = iris.data

plt.hist2d(x[:,0], x[:,1], cmap='Blues')
plt.xlabel('longueur sépal')
plt.ylabel('largeur sépal')
plt.colorbar()
```

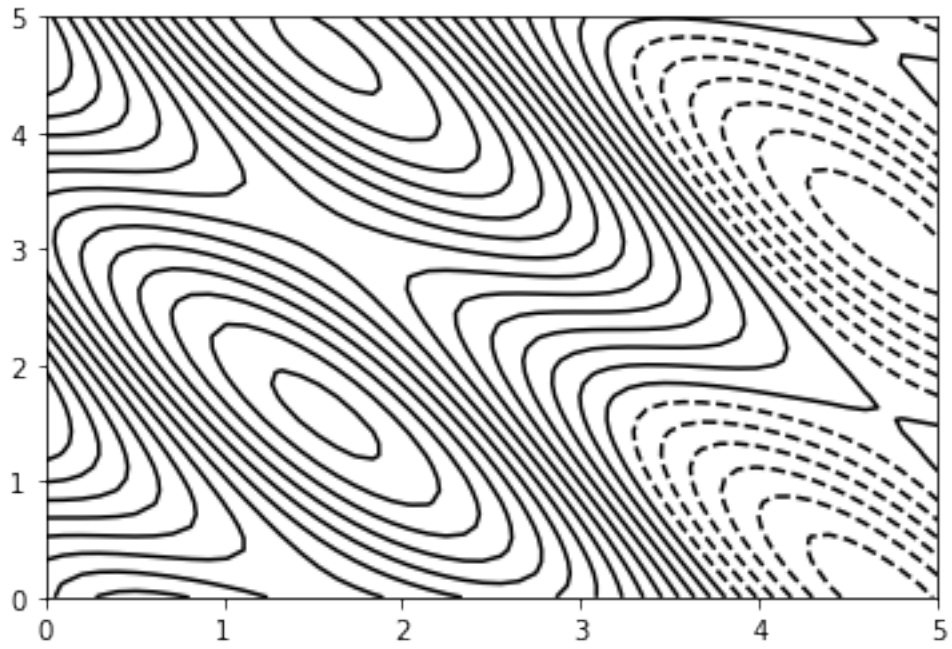
[12]: <matplotlib.colorbar.Colorbar at 0x7ff393c2b8d0>



8.3.4 Graphiques ContourPlot()

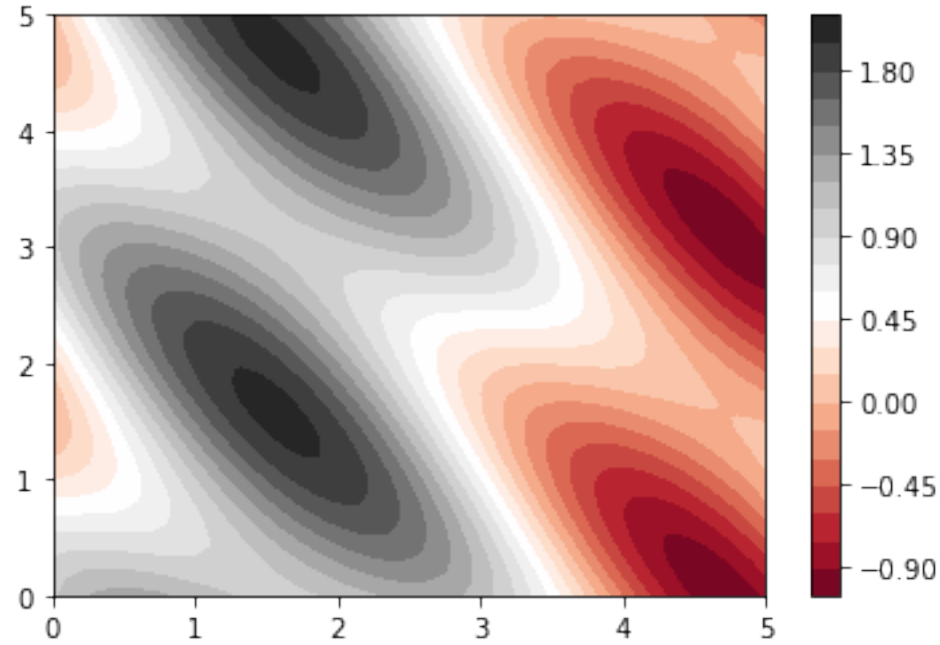
[31]:
```
f = lambda x, y: np.sin(x) + np.cos(x+y)*np.cos(x+y)

X = np.linspace(0, 5, 50)
Y = np.linspace(0, 5, 50)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)

plt.contour(X, Y, Z, 20, colors='black')
```

[31]:
```
<matplotlib.contour.QuadContourSet at 0x7fbde0e5f0a0>
```



[32]:
```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar()
```

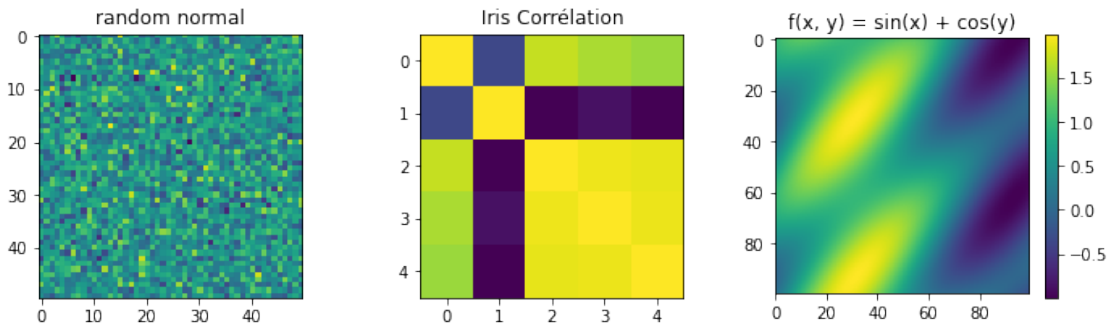
[32]:
```
<matplotlib.colorbar.Colorbar at 0x7fbde238a250>
```



8.3.5 Imshow()

[33]:
```
plt.figure(figsize=(12, 3))

# Simple graphique imshow()
X = np.random.randn(50, 50)

plt.subplot(131)
plt.imshow(X)
plt.title('random normal')

# Matrice de corrélation des iris
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target


plt.subplot(132)
plt.imshow(np.corrcoef(X.T, y))
plt.title('Iris Corrélation')

# Matrice f(X, Y) = sin(X) + cos(Y)
X = np.linspace(0, 5, 100)
Y = np.linspace(0, 5, 100)
X, Y = np.meshgrid(X, Y)

plt.subplot(133)
plt.imshow(f(X, Y))
plt.colorbar()
plt.title('f(x, y) = sin(x) + cos(y)')
```

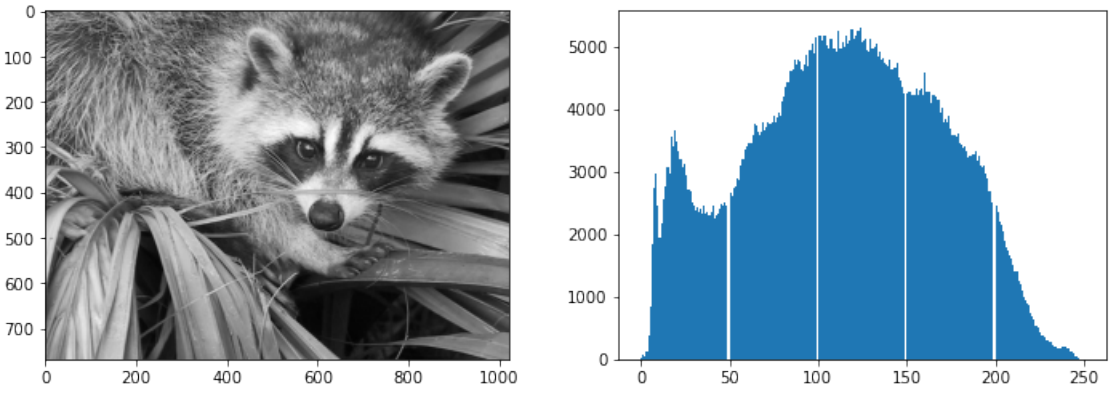
[33]:
```
Text(0.5, 1.0, 'f(x, y) = sin(x) + cos(y)')
```



8.4 Exercice

1. Ecrire un programme qui dessine une grille de graphiques 2X2 avec différentes largeurs de lignes, couleurs et styles.
2. Dans cet exercice, nous voulons afficher l'image ainsi que l'histogramme de l'échelle de gris comme indiqué sur la figure.

[22]:
```
# histogramme d'une image
from scipy import misc
face = misc.face(gray=True)

plt.figure(figsize=(12, 4))
plt.subplot(121)
# votre code ici
plt.subplot(122)
# votre code ici NB: utiliser la fonction reval()
plt.show()
```



9 TP 9: Pandas et Seaborn

9.1 Pandas

pandas est une bibliothèque logicielle écrite pour le langage de programmation Python pour la manipulation et l'analyse de données. En particulier, il propose des structures de données et des opérations de manipulation de tableaux numériques et de séries chronologiques.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
plt.style.use('ggplot')
```

9.1.1 DataFrame Pandas

Données tabulaires bidimensionnelles, variables en taille et potentiellement hétérogènes. La structure de données contient également des axes étiquetés (lignes et colonnes). Les opérations arithmétiques s'alignent sur les étiquettes de ligne et de colonne. Peut être considéré comme un conteneur de type dict pour les objets Series. La structure de données principale des pandas

9.1.2 Charger vos données dans un

Les options les plus courantes : - read_csv - read_excel

[3]:
```
data = pd.read_excel('Datasets/titanic3.xls')
```

[4]:
```
data.shape
```

[4]: (1309, 14)

9.1.3 Méthodes de Manipulation de Données avec Pandas

Lorsque vous travaillez avec des données tabulaires en Python, généralement stockées dans un DataFrame Pandas, il est essentiel de comprendre comment explorer et extraire des informations à partir de ces données. Voici quelques-unes des méthodes couramment utilisées pour examiner un DataFrame `data` :

1. `data.head()`

La méthode `data.head()` est utilisée pour afficher les premières lignes d'un DataFrame. Par défaut, elle affiche les 5 premières lignes. Cela peut être utile pour avoir un aperçu rapide des données.

```
# Affiche les 5 premières lignes du DataFrame
data.head()
```

Si vous souhaitez afficher un nombre différent de lignes, vous pouvez spécifier le nombre en passant un argument à la méthode. Par exemple, `data.head(10)` affichera les 10 premières lignes.

2. `data.tail()`

La méthode `data.tail()` est similaire à `data.head()`, mais elle affiche les dernières lignes du DataFrame. Par défaut, elle affiche les 5 dernières lignes.

```
# Affiche les 5 dernières lignes du DataFrame
data.tail()
```

Tout comme avec `data.head()`, vous pouvez spécifier le nombre de lignes à afficher en passant un argument, par exemple, `data.tail(10)` affichera les 10 dernières lignes.

3. `data.sample()`

La méthode `data.sample()` est utilisée pour extraire un échantillon aléatoire de lignes à partir du DataFrame. Par défaut, elle extrait une seule ligne de manière aléatoire.

```
# Extrait une ligne aléatoire du DataFrame
data.sample()
```

Pour spécifier le nombre d'échantillons à extraire, vous pouvez passer un argument tel que `data.sample(10)` pour obtenir 10 lignes aléatoires.

[9]:
```
data.head()
data.head(20)
data.tail()
data.tail(20)
data.sample(10)
```

[9]:
```
     pclass  survived                                             name  \
685       3         0                          Bowen, Mr. David John "Dai"
238       1         1  Robert, Mrs. Edward Scott (Elisabeth Walton Mc...
409       2         0                               Fox, Mr. Stanley Hubert
284       1         1         Stone, Mrs. George Nelson (Martha Evelyn)
150       1         0                                Harrison, Mr. William
209       1         1                             Mock, Mr. Philipp Edmund
996       3         0                                   Markun, Mr. Johann
589       2         1  Wells, Mrs. Arthur Henry ("Addie" Dart Trevaskis)
633       3         0                           Andreasson, Mr. Paul Edvin
50        1         1  Cardeza, Mrs. James Warburton Martinez (Charlo...

        sex   age  sibsp  parch    ticket      fare        cabin embarked  \
685    male  21.0      0      0     54636   16.1000          NaN        S
238  female  43.0      0      1     24160  211.3375           B3        S
409    male  36.0      0      0    229236   13.0000          NaN        S
284  female  62.0      0      0    113572   80.0000          B28      NaN
150    male  40.0      0      0    112059    0.0000          B94        S
209    male  30.0      1      0     13236   57.7500          C78        C
996    male  33.0      0      0    349257    7.8958          NaN        S
589  female  29.0      0      2     29103   23.0000          NaN        S
633    male  20.0      0      0    347466    7.8542          NaN        S
50   female  58.0      0      1  PC 17755  512.3292  B51 B53 B55        C

    boat   body                     home.dest
685  NaN    NaN    Treherbert, Cardiff, Wales
238    2    NaN                 St Louis, MO
409  NaN  236.0                Rochester, NY
284    6    NaN               Cincinatti, OH
150  NaN  110.0                          NaN
209   11    NaN                New York, NY
996  NaN    NaN                          NaN
589   14    NaN          Cornwall / Akron, OH
633  NaN    NaN            Sweden Chicago, IL
50     3    NaN  Germantown, Philadelphia, PA
```

[10]:
```
data.describe() # Avoir une analyse statistique des donnees
```

[10]:
```
            pclass     survived          age        sibsp        parch  \
count  1309.000000  1309.000000  1046.000000  1309.000000  1309.000000
mean      2.294882     0.381971    29.881135     0.498854     0.385027
std       0.837836     0.486055    14.413500     1.041658     0.865560
min       1.000000     0.000000     0.166700     0.000000     0.000000
25%       2.000000     0.000000    21.000000     0.000000     0.000000
50%       3.000000     0.000000    28.000000     0.000000     0.000000
75%       3.000000     1.000000    39.000000     1.000000     0.000000
max       3.000000     1.000000    80.000000     8.000000     9.000000

              fare        body
count  1308.000000  121.000000
mean     33.295479  160.809917
std      51.758668   97.696922
min       0.000000    1.000000
25%       7.895800   72.000000
50%      14.454200  155.000000
75%      31.275000  256.000000
max     512.329200  328.000000
```

9.1.4 Nettoyer des Datasets

La fonction "drop" supprime des lignes ou des colonnes en spécifiant les noms d'étiquette et l'axe correspondant, ou en spécifiant directement les noms d'index ou de colonne.

[11]:
```
data = data.drop(['name', 'sibsp', 'parch', 'ticket', 'fare', 'cabin',␣
↪'embarked', 'boat', 'body', 'home.dest'], axis=1)
```

Remplir les valeurs vides (NA/NaN) à l'aide de la méthode spécifiée. Dans ce cas, les données vides sont remplies par des moyens.

[12]:
```
data2= data.fillna(data['age'].mean())
print(data2.describe())
data2.shape
```

```
            pclass     survived          age
count  1309.000000  1309.000000  1309.000000
mean      2.294882     0.381971    29.881135
std       0.837836     0.486055    12.883199
min       1.000000     0.000000     0.166700
25%       2.000000     0.000000    22.000000
50%       3.000000     0.000000    29.881135
75%       3.000000     1.000000    35.000000
max       3.000000     1.000000    80.000000
```

[12]:
```
(1309, 4)
```

Ou nous les éliminerons tous.

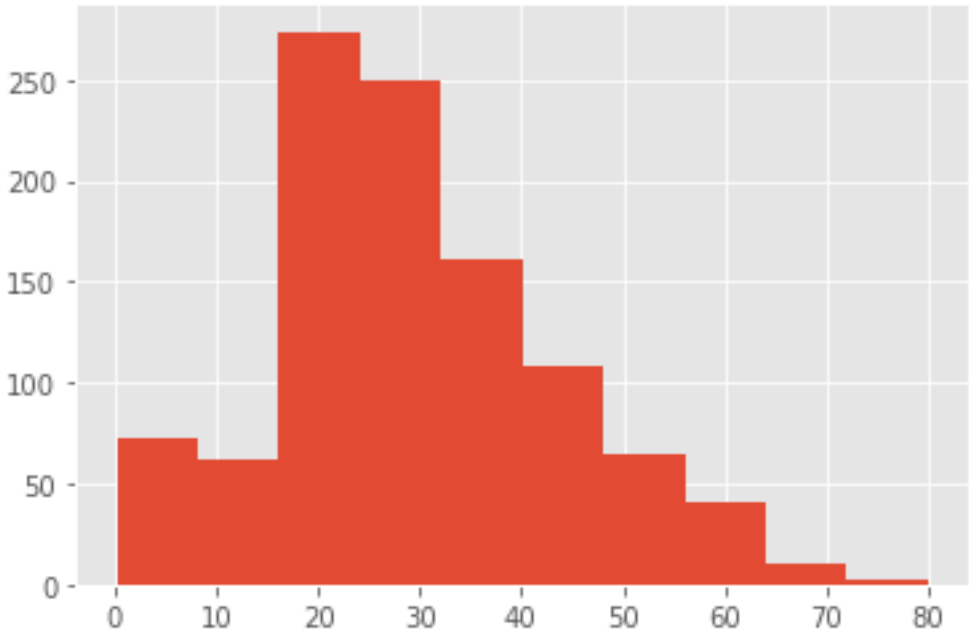
[13]:
```
data = data.dropna(axis=0)
print(data.describe())
data.shape
```

```
            pclass     survived          age
count  1046.000000  1046.000000  1046.000000
mean      2.207457     0.408222    29.881135
std       0.841497     0.491740    14.413500
min       1.000000     0.000000     0.166700
25%       1.000000     0.000000    21.000000
50%       2.000000     0.000000    28.000000
75%       3.000000     1.000000    39.000000
max       3.000000     1.000000    80.000000
```

[13]: (1046, 4)

[14]:
```
data['age'].hist()
```

[14]: <AxesSubplot:>



9.1.5 Statistics avec Groupby() et value_counts()

Une opération groupby implique une combinaison de fractionnement de l'objet, d'application d'une fonction et de combinaison des résultats. Cela peut être utilisé pour regrouper de grandes quantités de données et d'opérations de calcul sur ces groupes.

[15]:
```
data.groupby(['sex']).mean()
```

[15]:
```
          pclass  survived        age
sex
female  2.048969  0.752577  28.687071
male    2.300912  0.205167  30.585233
```

On observe que plus de femmes ont survécu au naufrage du Titanic.

[16]:
```
data.groupby(['sex', 'pclass']).mean()
```

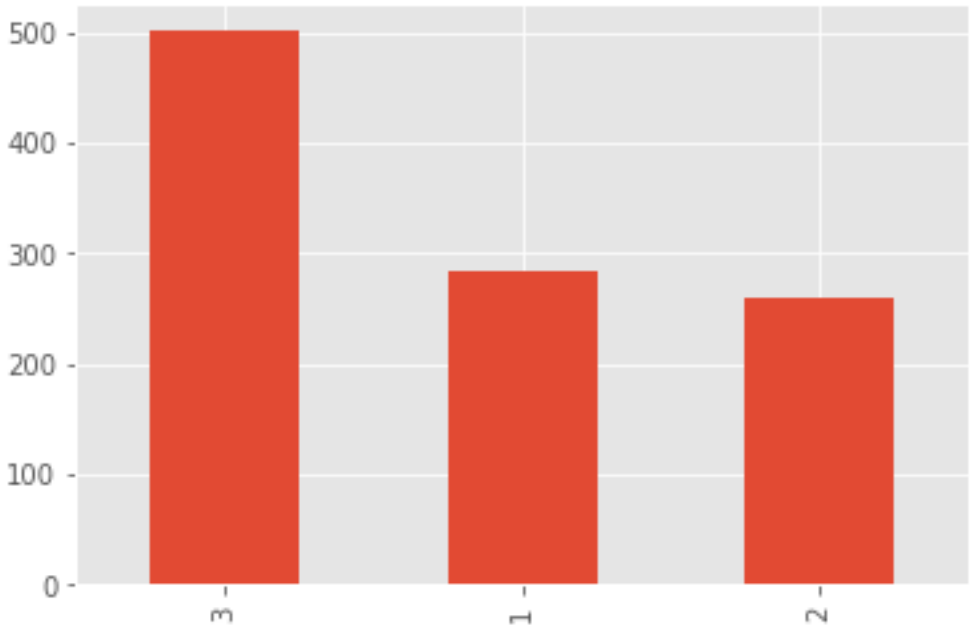
[16]:
```
               survived        age
sex    pclass
female 1       0.962406  37.037594
       2       0.893204  27.499191
       3       0.473684  22.185307
male   1       0.350993  41.029250
       2       0.145570  30.815401
       3       0.169054  25.962273
```

[17]:
```
data['pclass'].value_counts()
```

[17]:
```
3    501
1    284
2    261
Name: pclass, dtype: int64
```

[19]:
```
data['pclass'].value_counts().plot.bar()
```

[19]:
```
<AxesSubplot:>
```



Nous voulons obtenir le nombre de passagers par classe qui avaient moins de 18 ans.

[20]:
```
data[data['age'] < 18]['pclass'].value_counts()
```

[20]:
```
3    106
2     33
1     15
Name: pclass, dtype: int64
```

[21]:
```
data[data['age']>18].groupby(['sex','pclass']).mean()
```

[21]:
```
               survived        age
sex    pclass
female 1       0.966667  39.358333
       2       0.878049  32.067073
       3       0.436170  29.457447
male   1       0.328671  42.716783
       2       0.087591  34.069343
       3       0.158845  29.799639
```

9.1.6 Operation sur les serie

Ndarray unidimensionnel avec étiquettes d'axe (y compris les séries chronologiques).

Les étiquettes n'ont pas besoin d'être uniques, mais doivent être de type hachable. L'objet prend en charge l'indexation basée à la fois sur les entiers et sur les étiquettes et fournit une multitude de méthodes pour effectuer des opérations impliquant l'index. Les méthodes statistiques de ndarray ont

été remplacées pour exclure automatiquement les données manquantes (actuellement représentées par NaN).

[20]:
```
#data['age'] est une serie
print(data['age'][0:10]) # slicing a serie
data2 = data['age']> 18 # cree un mask
print(data2)
print(data[data['age']< 18]) # boolean indexing
```

```
0    29.0000
1     0.9167
2     2.0000
3    30.0000
4    25.0000
5    48.0000
6    63.0000
7    39.0000
8    53.0000
9    71.0000
Name: age, dtype: float64
0        True
1       False
2       False
3        True
4        True
        ...
1301     True
1304    False
1306     True
1307     True
1308     True
Name: age, Length: 1046, dtype: bool
      pclass  survived     sex      age
1          1         1    male   0.9167
2          1         0  female   2.0000
53         1         0    male  17.0000
54         1         1    male  11.0000
55         1         1  female  14.0000
...      ...       ...     ...      ...
1265       3         0  female  10.0000
1275       3         0    male  16.0000
1279       3         0  female  14.0000
1300       3         1  female  15.0000
1304       3         0  female  14.5000

[154 rows x 4 columns]
```

[52]:
```
data.head()
```

[52]:
```
   pclass  survived  sex      age
0       1         1    0  29.0000
1       1         1    1   0.9167
2       1         0    0   2.0000
3       1         0    1  30.0000
4       1         0    0  25.0000
```

9.1.7 Méthodes loc et iloc

Les méthodes loc et iloc sont des méthodes Pandas essentielles utilisées pour filtrer, sélectionner et manipuler des données. Ils nous permettent d'accéder à la combinaison souhaitée de lignes et de colonnes.

La principale différence entre eux est la façon dont ils accèdent aux lignes et aux colonnes :

- loc utilise des étiquettes de ligne et de colonne.
- iloc utilise des index de ligne et de colonne.

[60]:
```
print(data.iloc[3,3]) # iloc pour la localisation d'index récupère la valeur de␣
 ↪la 4ème ligne et de la 4ème colonne
print(data.iloc[0,3]) # récupère la valeur de la première ligne et de la 4ème␣
 ↪colonne
print(data.loc[0:2,'age'])
```

```
30.0
29.0
0    29.0000
1     0.9167
2     2.0000
Name: age, dtype: float64
```

9.1.8 Codefication des donnees

Dans cette partie nous allons créer des catégories et codifier les donnees string avec les fonctions map(), replace() et cat.codes.

[12]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

[46]:
```
data = pd.read_excel('Datasets/titanic3.xls')
data = data.drop(['name', 'sibsp', 'parch', 'ticket', 'fare', 'cabin',␣
 ↪'embarked', 'boat', 'body', 'home.dest'], axis=1)
data = data.dropna(axis=0)
```

[41]:
```
data.head()
```

[41]:
```
   pclass  survived     sex      age
0       1         1  female  29.0000
```

```
1       1         1    male   0.9167
2       1         0  female   2.0000
3       1         0    male  30.0000
4       1         0  female  25.0000
```

Pour créer des catégories d'âge comme indiqué dans l'exemple, il existe deux méthodes, nous utilisons soit l'indexation boléenne, soit la fonction de carte qui est illustrée ci-dessous

[ ]:
```
data.loc[data['age']< 20,'age']=0
data.loc[(data['age']>= 20)  & ( data['age']< 30),'age']=1
data.loc[(data['age']>= 30)  &  (data['age']< 40),'age']=2
data.loc[data['age']>= 40,'age']=3
data.head()
```

[17]:
```
data['age'].value_counts()
```

[17]:
```
1.0    344
3.0    245
2.0    232
0.0    225
Name: age, dtype: int64
```

[18]:
```
data.groupby(['age']).mean()
```

[18]:
```
       pclass  survived
age
0.0  2.542222  0.471111
1.0  2.436047  0.369186
2.0  2.103448  0.422414
3.0  1.677551  0.391837
```

La fonction map() applique une fonction sur tous les elements d'une colone.

[34]:
```
data['age'].map(lambda x:x+2)
```

[34]:
```
0       31.0000
1        2.9167
2        4.0000
3       32.0000
4       27.0000
         ...
1301    47.5000
1304    16.5000
1306    28.5000
1307    29.0000
1308    31.0000
Name: age, Length: 1046, dtype: float64
```

[31]:
```
def category_ages(age):
    if age <= 20:
        return '<20 ans'
    elif (age > 20) & (age <= 30):
        return '20-30 ans'
    elif (age > 30) & (age <= 40):
        return '30-40 ans'
    else:
        return '+40 ans'
```

[35]:
```
data['age'] = data['age'].map(category_ages)
```

[36]:
```
data.head()
```

[36]:

	pclass	survived	sex	age
0	1	1	female	20-30 ans
1	1	1	male	<20 ans
2	1	0	female	<20 ans
3	1	0	male	20-30 ans
4	1	0	female	20-30 ans

De plus, ici, la colonne sex contient des informations au format String, ce qui est inhabituel et inadapté à une application d'apprentissage automatique. Ainsi, nous utiliserons une codification pour convertir cette colonne au format numérique. Pour cette tâche, il existe trois méthodes adaptées à cette tâche : la fonction map(), replace() et cat.codes

[38]:
```
def Conv_sex_to_number(sex):
    if sex == 'female':
        return 0
    else :
     return 1
data['sex'] = data['sex'].map(Conv_sex_to_number)
data.head()
```

[38]:

	pclass	survived	sex	age
0	1	1	0	20-30 ans
1	1	1	1	<20 ans
2	1	0	0	<20 ans
3	1	0	1	20-30 ans
4	1	0	0	20-30 ans

[43]:
```
data['sex'] = data['sex'].map({'male':0,'female':1})
data.head()
```

[43]:

	pclass	survived	sex	age
0	1	1	1	1.0
1	1	1	0	0.0

```
2       1         0    1  0.0
3       1         0    0  2.0
4       1         0    1  1.0
```

[45]:
```
data['sex'] = data['sex'].replace(['male','female'],[0,1])
data.head()
```

[45]:
```
   pclass  survived  sex      age
0       1         1    1  29.0000
1       1         1    0   0.9167
2       1         0    1   2.0000
3       1         0    0  30.0000
4       1         0    1  25.0000
```

[51]:
```
data['sex']= data['sex'].astype('category').cat.codes
data.head()
```

[51]:
```
   pclass  survived  sex      age
0       1         1    0  29.0000
1       1         1    1   0.9167
2       1         0    0   2.0000
3       1         0    1  30.0000
4       1         0    0  25.0000
```

[ ]:
```

```

9.2 Seaborn

Seaborn est une bibliothèque Python de visualisation de données basée sur matplotlib. Elle fournit une interface de haut niveau pour dessiner des graphiques statistiques attrayants et informatifs.

Vous pouvez en savoir plus sur Seaborn : https://seaborn.pydata.org/api.html

[4]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
plt.style.use('ggplot')
```

[5]:
```
iris = sns.load_dataset('iris')
iris.head()
```

[5]:
```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

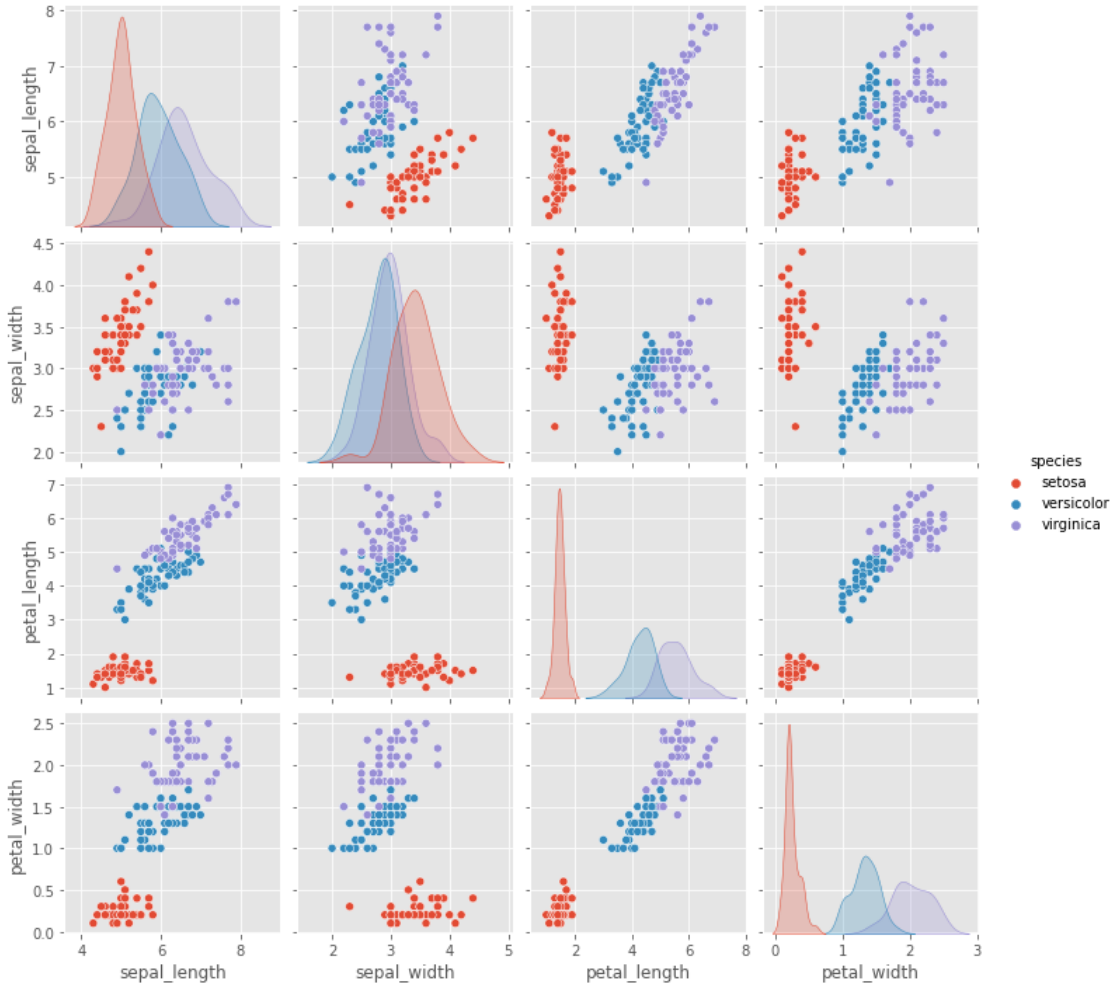
9.2.1 La vue d'ensemble Pairplot()

Tracer des relations par paires dans un jeu de données. Par défaut, cette fonction créera une grille d'axes telle que chaque variable numérique dans les données sera partagée sur les axes y sur une seule ligne et les axes x sur une seule colonne. Les diagrammes diagonaux sont traités différemment : un diagramme de distribution univariée est tracé pour montrer la distribution marginale des données dans chaque colonne.

Il est également possible d'afficher un sous-ensemble de variables ou de tracer différentes variables sur les lignes et les colonnes.

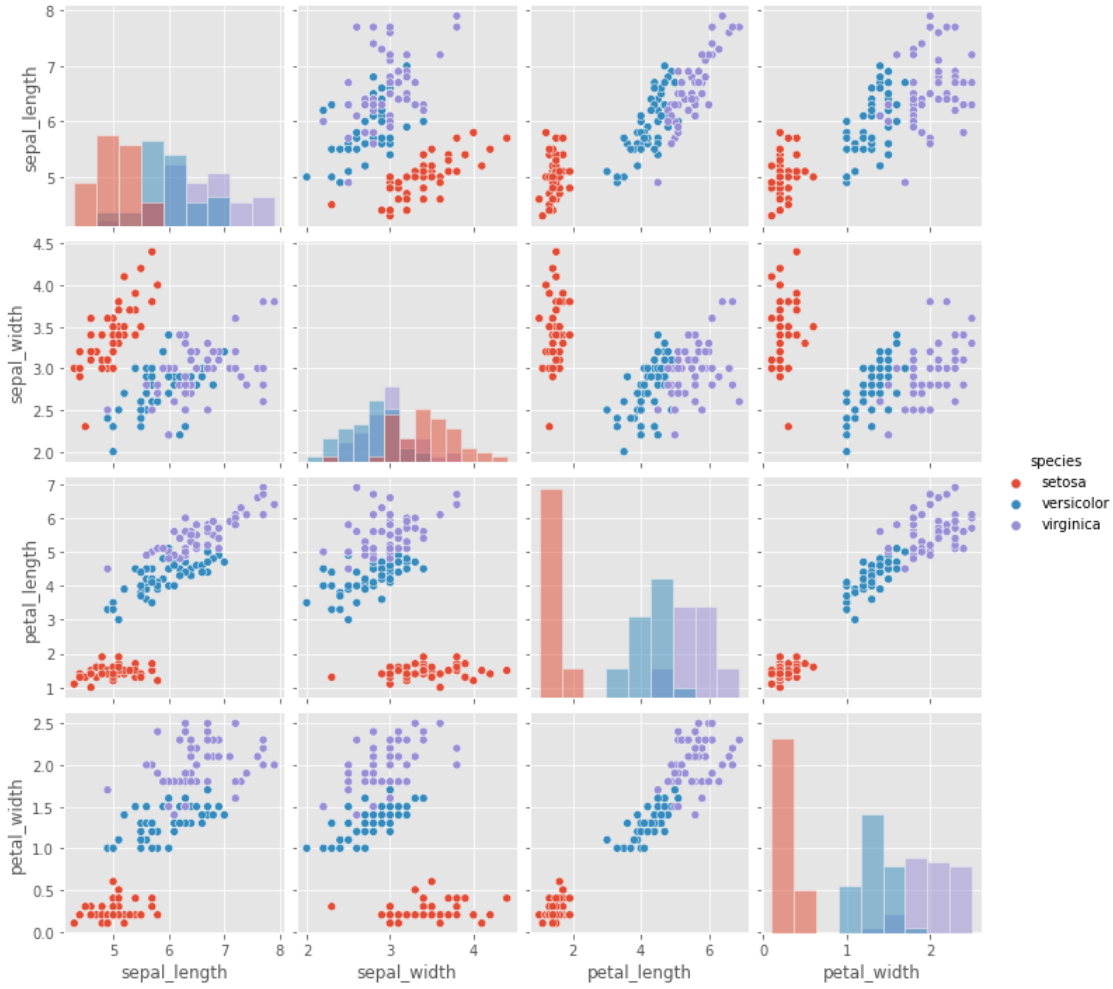
[6]:
```
sns.pairplot(iris, hue='species')
```

[6]:
```
<seaborn.axisgrid.PairGrid at 0x7f9ac50bf1c0>
```



[7]:
```
sns.pairplot(iris, hue='species', diag_kind="hist")
```

[7]:
```
<seaborn.axisgrid.PairGrid at 0x7f9aca5eb130>
```



9.2.2 Visualiser de catégories

Interface au niveau de la figure pour dessiner des tracés catégoriels sur un FacetGrid.

Cette fonction donne accès à plusieurs fonctions au niveau des axes qui montrent la relation entre une variable numérique et une ou plusieurs variables catégorielles à l'aide d'une des nombreuses représentations visuelles.

[9]:
```
titanic = sns.load_dataset('titanic')
titanic.drop(['alone', 'alive', 'who', 'adult_male', 'embark_town', 'class'],␣
 ↪axis=1, inplace=True)
titanic.dropna(axis=0, inplace=True)
titanic.head()
```

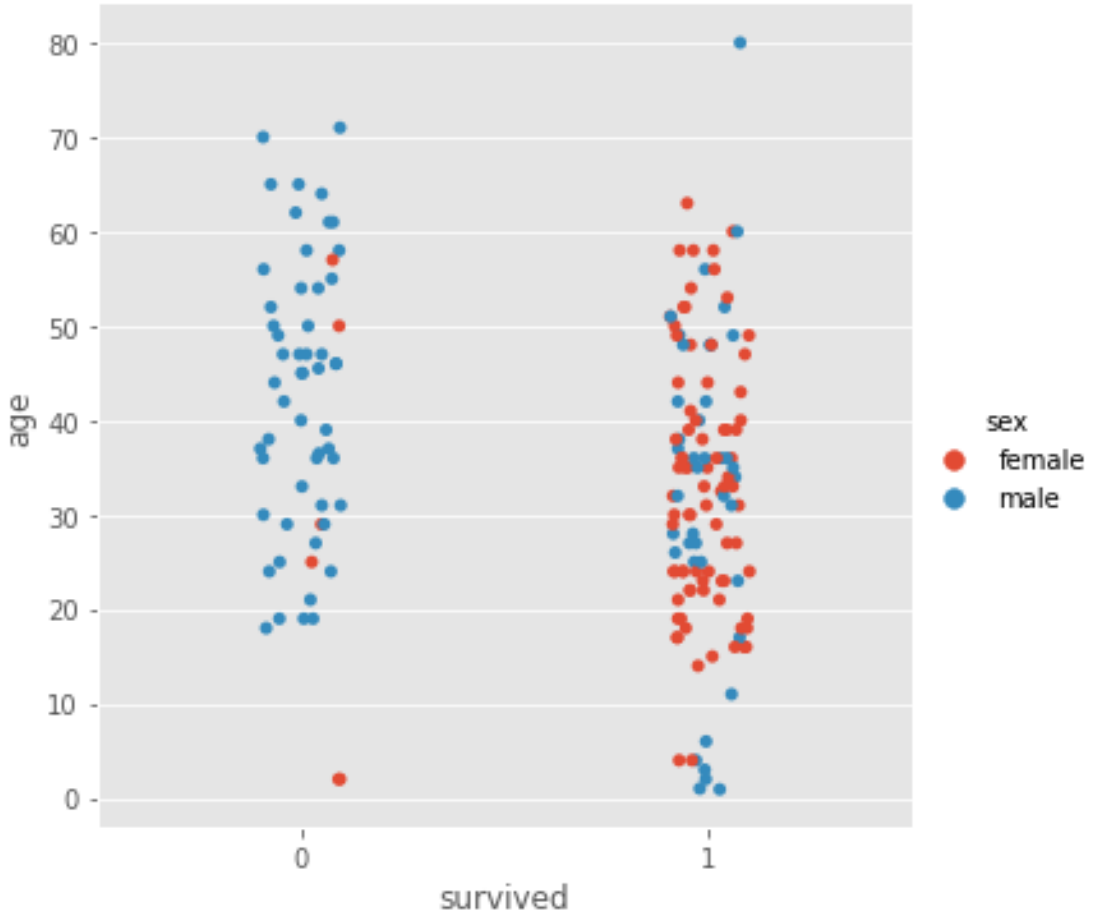
[9]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	deck
1	1	1	female	38.0	1	0	71.2833	C	C
3	1	1	female	35.0	1	0	53.1000	S	C

```
6          0       1    male  54.0      0      0  51.8625        S    E
10         1       3  female   4.0      1      1  16.7000        S    G
11         1       1  female  58.0      0      0  26.5500        S    C
```

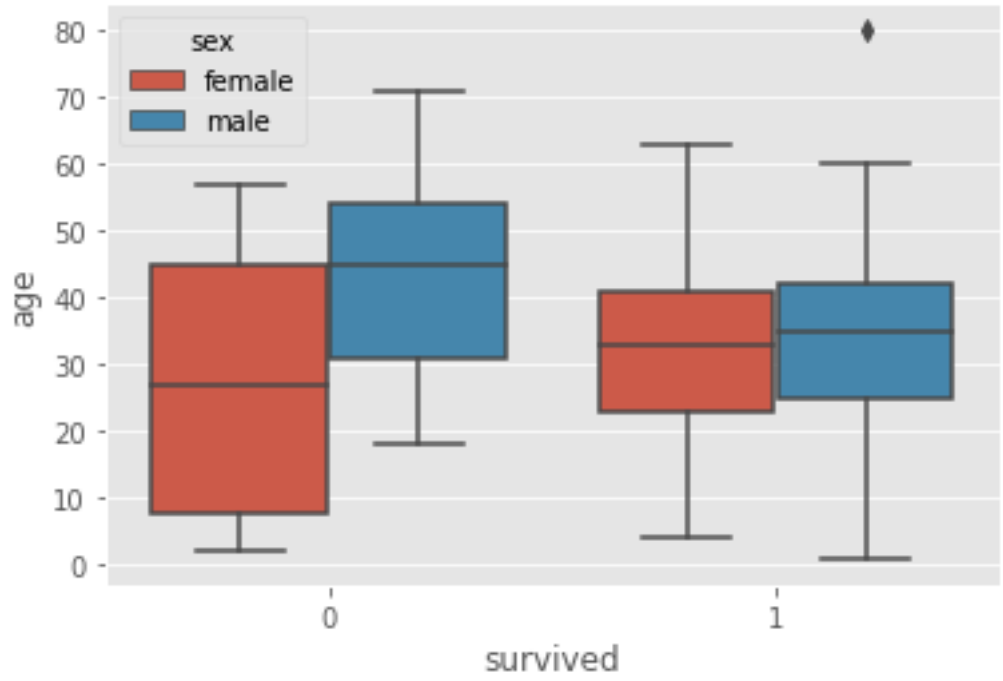
[10]:
```
sns.catplot(x='survived', y='age', data=titanic, hue='sex')
```

[10]:
```
<seaborn.axisgrid.FacetGrid at 0x7f9aca580b50>
```



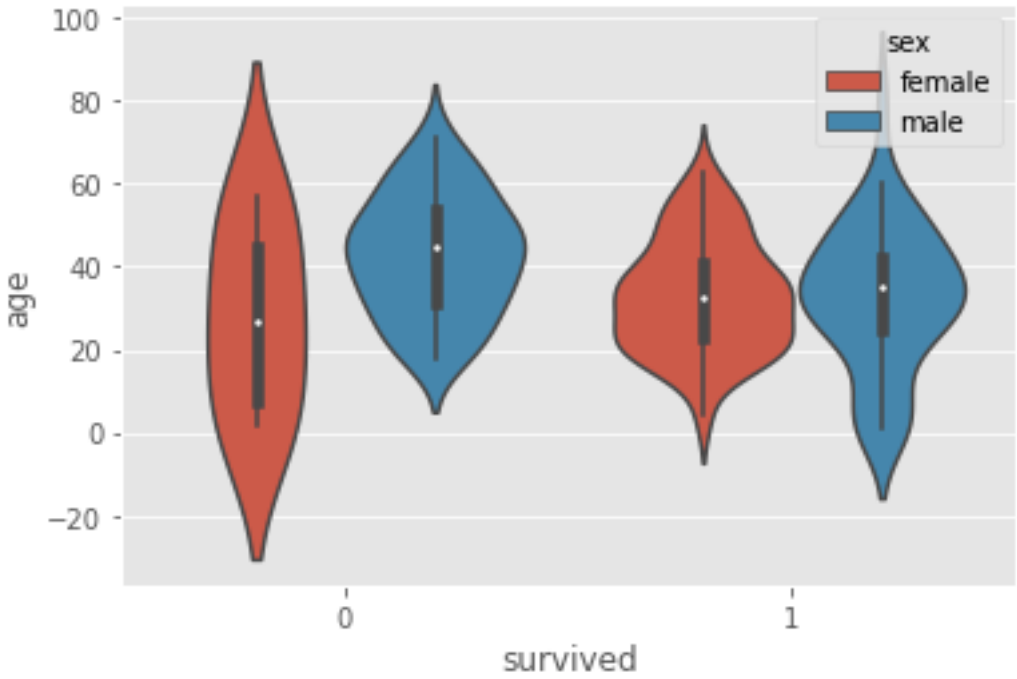
[12]:
```
sns.boxplot(x='survived',y='age', data=titanic,hue='sex')
```

[12]:
```
<AxesSubplot:xlabel='survived', ylabel='age'>
```

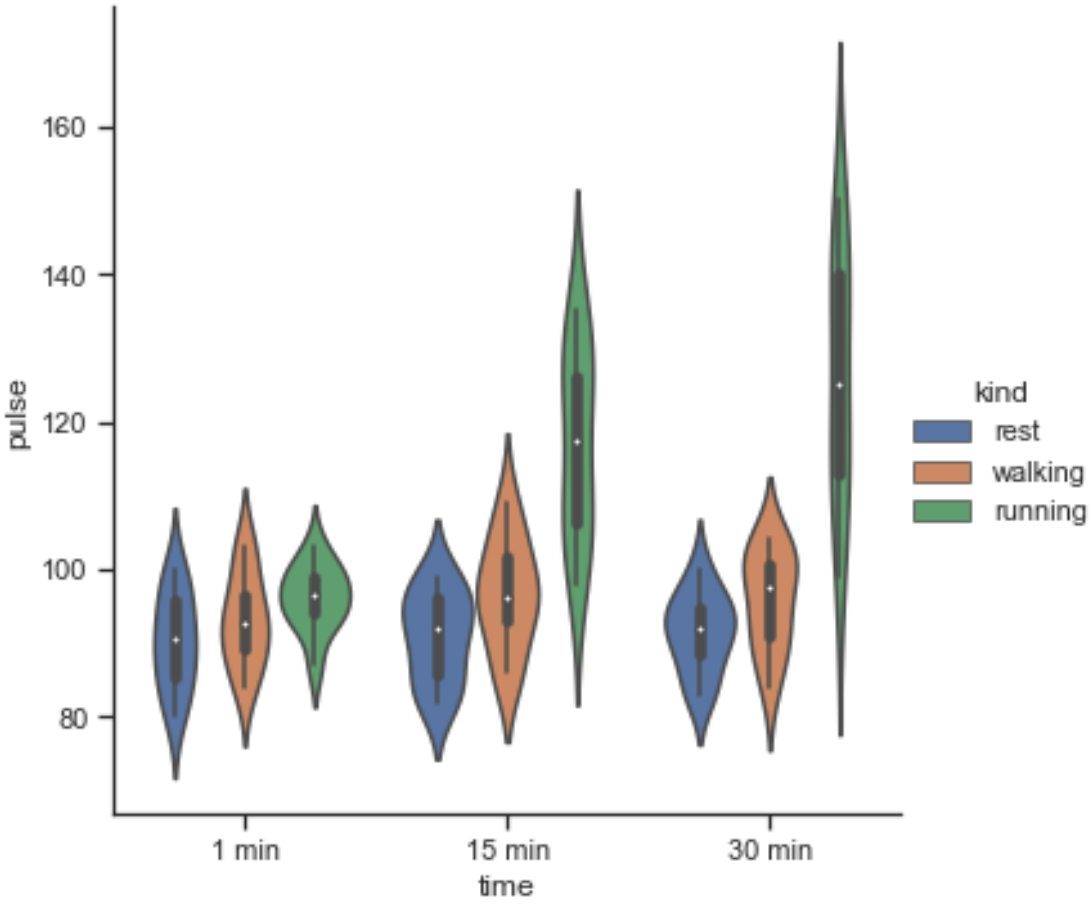


[13]:
```
sns.violinplot(x='survived',y='age', data=titanic,hue='sex')
```

[13]:
```
<AxesSubplot:xlabel='survived', ylabel='age'>
```

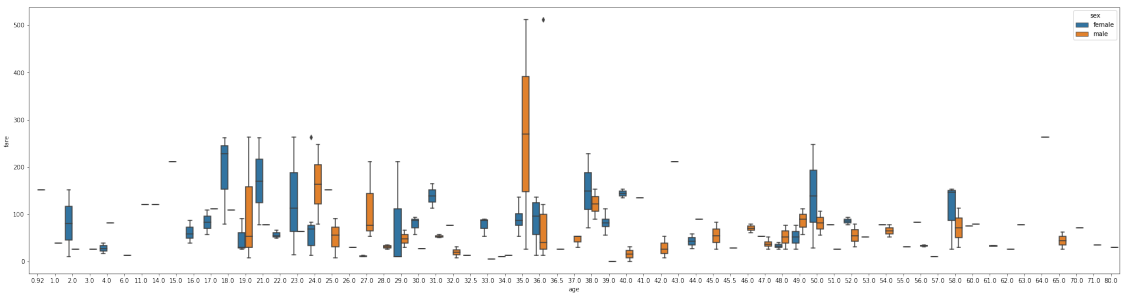


[22]:
```
sns.set_theme(style="ticks")
exercise = sns.load_dataset("exercise")
g = sns.catplot(x="time", y="pulse", hue="kind",data=exercise, kind="violin")
```



[8]:
```
plt.figure(figsize=(32, 8))
sns.boxplot(x='age', y='fare', data=titanic, hue='sex')
```

[8]:
```
<AxesSubplot:xlabel='age', ylabel='fare'>
```

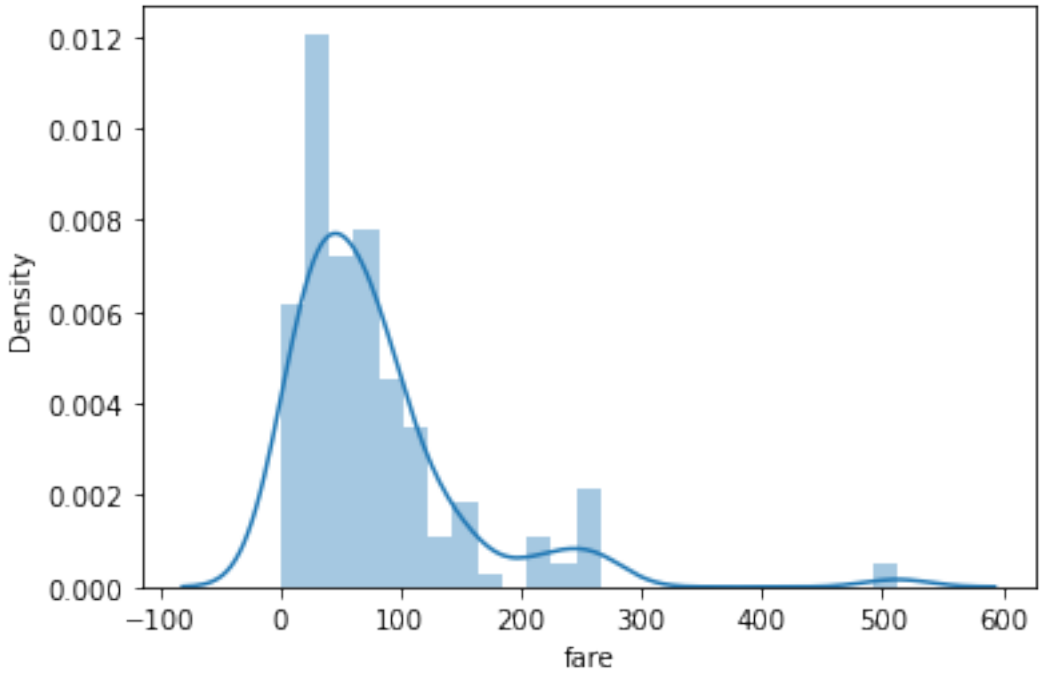


9.2.3 Visualisation de Distributions

[13]:
```
sns.distplot(titanic['fare'])
```

```
/Users/mac/opt/anaconda3/lib/python3.8/site-
packages/seaborn/distributions.py:2551: FutureWarning: `distplot` is a
deprecated function and will be removed in a future version. Please adapt your
code to use either `displot` (a figure-level function with similar flexibility)
or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

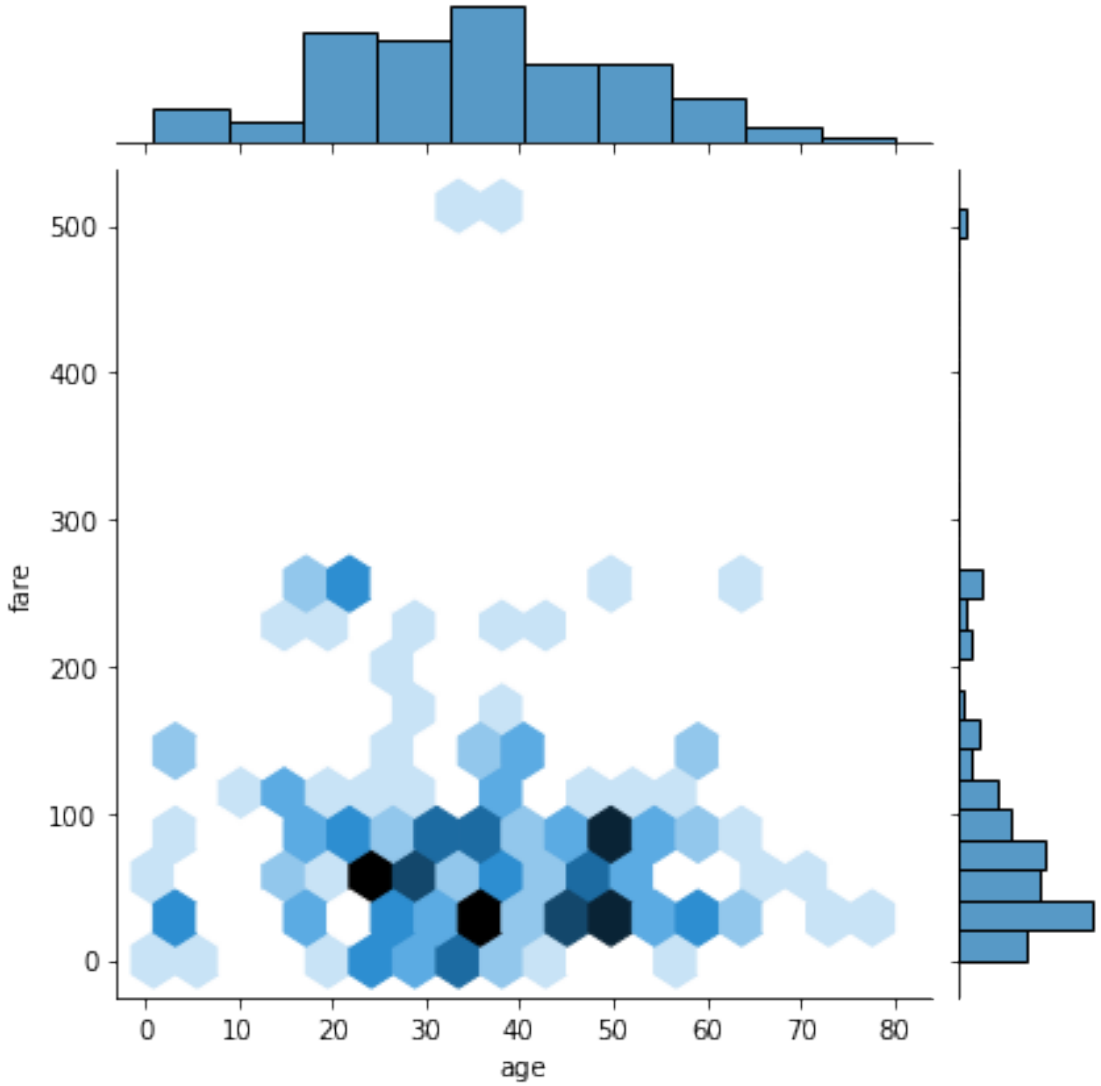
[13]:
```
<AxesSubplot:xlabel='fare', ylabel='Density'>
```



[14]:
```
sns.jointplot('age', 'fare', data=titanic, kind='hex')
```

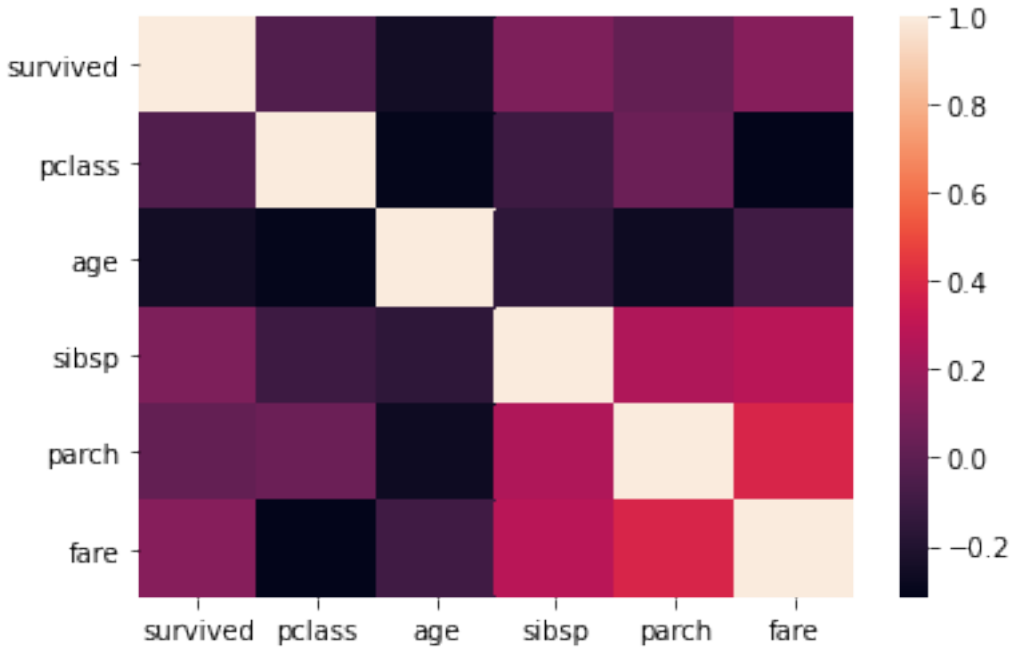
```
/Users/mac/opt/anaconda3/lib/python3.8/site-packages/seaborn/_decorators.py:36:
FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  warnings.warn(
```

[14]: <seaborn.axisgrid.JointGrid at 0x7f9c3b8d6ac0>



[15]:
```
sns.heatmap(titanic.corr())
```

[15]: <AxesSubplot:>



10 TP 10: Expressions régulière

Pour utiliser les expressions régulières sous `Python` il faut importer le module `re` :

[1]:
```
import re
```

Une expression régulière est d'abord compilée et le résultat est stocké dans un objet `RegexObject`. On écrit une expression régulière dans une «chaîne brute Python» : une chaîne délimitée par `r"` et `"`. Exemple :

[2]:
```
reg= r"[a-z]+" # est l'expression régulière qui correspond aux chaînes formées␣
↪d'une ou plusieurs lettres entre a et z.
reg
```

[2]:
```
'[a-z]+'
```

Pour compliler cette expression et créer un objet `RegexObject` on écrit :

[3]:
```
reg = re.compile(reg)
reg
```

[3]:
```
re.compile(r'[a-z]+', re.UNICODE)
```

L'objet `reg` a plusieurs méthodes, nous allons en utiliser trois :

1. `findall` qui sert à appliquer l'expression régulière et à récupérer les sous-chaînes trouvées sous forme de liste Python ;
2. `finditer` qui sert à appliquer l'expression régulière et à récupérer les sous-chaînes trouvées sous forme d'itérateur Python ;
3. `sub` qui sert à appliquer une expression régulière, à remplacer les sous-chaînes trouvées par d'autres chaînes.

Il s'agit donc de fonctionnalités similaires aux «chercher» et «chercher / remplacer» des éditeurs de texte. Une méthode plus simple, du nom de `match()` va simplement tester si une chaîne satisfait les contraintes imposées par l'objet expression régulière auquel on l'applique.

10.1 Syntaxe des expressions régulières

Avant de voir l'utilisation des expressions régulières sous Python, un rappel de la syntaxe des expressions régulières «à la Perl» :

- `toto` va trouver les sous-chaînes `toto` ;
- . est un caractère quelconque, mis à part le passage à la ligne `\n` et le retour chariot `\r` ;
- [ax123Z] signifie : `un caractère quelconque parim a, x, 1, 2, 3 et Z` ;
- [A-Z] signifie : `un caractère quelconque dans l'intervalle de A à Z` ;
- le trait d'union sert à indiquer les intervalles mais peut faire partie des caractères recherchés s'il est placé à la fin : `[AZ-]` signifie : `un caractère quelconque parmi A, Z et -` ;
- on peut combiner à volonté les caractères énumérés et les intervalles : par exemple `[A-Za-z0-9.:?]` signifie `une lettre minuscule ou majuscule, un chiffre, un point, un deux-points, ou un point d'interrogation` ;
- les caractères (,), , [,] peuvent être recherchés, à condition de les protéger par un antislash : `\(, \), \\, \[, \]` ;
- le symbole `^` placé après le crochet ouvrant indique que l'on va chercher le complémentaire de ce qui est placé entre les crochets. Exemple : `[^a-z]` va trouver un caractère quelconque qui ne soit pas une lettre entre a et z ;
- on dispose des quantificateurs suivants :
  1. `*` (zéro, une ou plusieurs fois),
  2. `+` (une ou plusieurs fois),
  3. `?` (zéro ou une fois),
  4. `{n,m}` (entre n et m fois),
  5. `{n,}` (plus de n fois) ;
- on dispose également des quantificateurs «non gourmands» suivants :
  1. `*?` (zéro, une ou plusieurs fois),
  2. `+?` (une ou plusieurs fois),
  3. `??` (zéro ou une fois),
  4. `{n,m}?` (entre n et m fois),
  5. `{n,}?` (plus de n fois) ;

La différence entre quantificateurs «gourmands» et «non gourmands» provient du fait que les premiers vont trouver la sous-chaîne la plus longue respectant les contraintes alors que les deuxièmes vont trouver la chaîne la plus courte.

1. Exemple : l'expression `[a-z]+` appliquée à `«mon ami Pierrot»` va trouver `mon`, alors que `[a-z]+?` va trouver `m` (ce qui n'a que peu d'intérêt).

2. Autre exemple (qui montre l'utilité des quantificateurs non gourmands) : l'expression `\(.+\)` appliquée à «Brest (29) et Aix (13)» va retourner 29) et Aix (13 puisque c'est la plus longue sous-chaîne délimitée par une parenthèse ouvrante et une parenthèse fermante. Par contre (.+?) va retourner d'abord 29 et ensuite 13

- les symboles `^` et `$` servent à indiquer le début et la fin d'une chaîne. Par exemple : `^a.+` va trouver toutes les chaînes qui commencent par un a, `toto$` va trouver toutes les chaînes qui finissent par toto, `^ $` va trouver toutes les chaînes égales à un blanc ;
- l'opérateur «ou» `|` sert à indiquer un choix entre deux expressions ;
- on peut utiliser les parenthèses pour deux raisons :
  1. pour délimiter une expression qui sera utilisée par l'opérateur «ou» ou à laquelle on va appliquer un quantificateur (exemple : abc(toto)+ signifie «abc suivi d'un ou plusieurs toto») ;
  2. pour délimiter une sous-chaîne que l'on va récupérer par la suite. On appelle cette sous-chaîne, un «groupe».

Ce double usage des parenthèses peut être gênant : en écrivant `abc(toto)+` on fait de toto un groupe, même si on n'a pas l'intention de le récupérer par la suite. En écrivant `abc(?:toto)+` les parenthèses ne servent qu'au premier usage, aucun groupe n'est formé.

10.2 Utilisation des expressions régulières sous Python

10.2.1 Recherche

Supposons que l'on veuille trouver tous les mots de la chaîne «Le bon chasseur sachant chasser sait chasser sans son chien» contenant un «s». On peut trouver un tel mot en écrivant `[a-rt-z]*s[a-z]*`. On peut donc déjà compiler une expression régulière :

[4]:
```
r = re.compile(r"[a-rt-z]*s[a-z]*")
```

Pour l'appliquer à la chaîne on écrira :

[5]:
```
m = r.findall("Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

```
['chasseur', 'sachant', 'chasser', 'sait', 'chasser', 'sans', 'son']
```

On peut ré-écrire le code précédent en utilisant un `itérateur` Python :

[6]:
```
for m in r.finditer("Le bon chasseur sachant chasser sait chasser sans son␣
↪chien"):
    print(m.group())
```

```
chasseur
sachant
chasser
sait
chasser
sans
son
```

L'avantage de cette écriture est que l'on récupère non pas des simples chaînes de caractères mais des objets MatchObject qui ont leurs propres méthodes et attributs.

10.2.2 Recherche / remplacement

Maintenant nous allons essayer de rendre la phrase «Le bon chasseur sachant chasser sait chasser sans son chien» conforme au dialecte chti-mi. On peut commencer par remplacer tous les «s» par des «ch» :

[7]:
```
r = re.compile(r"s")
m = r.sub(r"ch","Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

```
Le bon chachcheur chachant chachcher chait chachcher chanch chon chien
```

Le résultat est «Le bon chachcheur chachant chachcher chait chachcher chanch chon chien», qui est relativement imprononçable.

On peut rectifier le tir en évitant les doubles «ch». On va donc remplacer un nombre quelconque de lettres s consécutives par un seul «ch» : import re

[8]:
```
r = re.compile(r"s+")
m = r.sub(r"ch","Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

```
Le bon chacheur chachant chacher chait chacher chanch chon chien
```

Le résultat «Le bon chacheur chachant chacher chait chacher chanch chon chien» est nettement plus chti-mi, mais il reste un cas problématique : le «s» muet du mot «sans» est devenu un «ch» prononcé dans «chanch». Il faut donc éviter de convertir les «s» en fin de mot :

[9]:
```
r = re.compile(r"s+([a-z]+)")
m = r.sub(r"ch\1","Le bon chasseur sachant chasser sait chasser sans son chien")
print(m)
```

```
Le bon chacheur chachant chacher chait chacher chans chon chien
```

Pour ce faire, on a créé un groupe ([a-z]+) que l'on retrouve dans la chaîne de remplacement (\1). Le réchultat «Le bon chacheur chachant chacher chait chacher chans chon chien» est chans contechte parfaitement chti-mi.

10.2.3 Recherche / remplacement avec utilisation de fonction

Lorsqu'on remplace une sous-chaîne par une autre il peut être utile d'intercaler un traitement entre lecture de la sous-chaîne et écriture dans la nouvelle chaîne. Python nous permet d'appliquer une fonction à chacune des sous-chaînes trouvées. Imaginons que dans la chaîne `toto 123 blabla 456 titi` on veut représenter les nombres en hexadécimal. Ce calcul est trop compliqué pour être fait uniquement par des expressions régulières, on utilisera donc une fonction

[10]:
```
def ecrire_en_hexa ( entree ):
    return hex( int( entree.group() ) )
```

```
r = re.compile(r"[0-9]+")
m = r.sub( ecrire_en_hexa,"toto 123 blabla 456 titi" )
print(m)
```

```
toto 0x7b blabla 0x1c8 titi
```

Le résultat est bien `toto 0x7b blabla 0x1c8 titi`. L'argument de la fonction est un objet de type `MatchObject`. La méthode group() fournit la chaîne tout entière, alors que group(n) fournira le n-ième groupe de la sous-chaîne.

11 Exercices

Lire le fichier Personnels.csv. Pour le lire ligne par ligne, utiliser le code Python suivant :

[12]:
```
f = open("Personnels.csv",'r')
for ligne in f:
   pass;
        #faire qqch avec la ligne ligne
f.close()
```

11.1 EXO 1 :

Que fait le code suivant ?

[11]:
```
r = re.compile(r"^([0-9]+);[^;]*;MALIK;")
f = open("Personnels.csv",'r')
for ligne in f:
    for m in r.finditer(ligne):

        print(m.group(1)+" OK")
f.close()

# Votre reponse en commentaire
```

```
1 OK
4 OK
5 OK
```

11.2 EXO2 :

Complétez ce programme afin qu'il sorte les identifiants des gens nés dans un village dont le nom commence par AIN.

[13]:
```
r = re.compile(r"^([0-9]+);[^;]*;MALIK;[^;]*;AIN")
f = open('Personnels.csv','r')
for ligne in f:
```

```
    for m in r.finditer(ligne):
        print(m.group(1)+" OK")
f.close()
```

```
1 OK
4 OK
5 OK
```

Remplacez les lieux de naissance des personnes trouvées dans l'exercice 2 par des lieux qui commencent par BEN- (par exemple : AINBAIDA devient BEN-BAIDA). Rappel : pour écrire dans un fichier on utilise le code suivant :

[14]:
```
o = open("fichier_sortie.csv",'w')
o.write("texte à écrire")
o.close()
```

[15]:
```
r = re.compile(r"^([0-9]+;[^;]*;MALIK;[^;]*;)AIN")
f = open('Personnels.csv','r')
o = open("fichier_sortie.csv",'w')
for ligne in f:
    for m in r.finditer(ligne):
        o.write(r.sub(r"\1BEN-",ligne))
f.close()
o.close()
```

11.3 EXO 3

Incrémentez les dates de naissance des personnes trouvées dans l'exercice 2 de 10 ans.

Tips : 1. Définir une fonction traiterdate qui va gérer le remplacement de la chaîne qui nous intéresse ; 2. En Python on passe d'un objet nombre entier à un objet chaîne en utilisant str, pour l'opération inverse on dispose de la fonction int.

[ ]:
```
import re
def traiterdate( entree ):
    return entree.group(1)+"/"+str(int(entree.group(2))+10)+";AIN"
r = re.compile(r"^([0-9]+;[^;]*;MALIK;[0-9][0-9]/[0-9][0-9])/([0-9]{4});AIN")
f = open('Personnels.csv','r')
o = open("fichier_sortie.csv",'w')
for ligne in f:
    o.write(r.sub(traiterdate,ligne))
f.close()
o.close()
```

11.4 EXO 4

Calculez l'âge moyen lors des mariages, en considérant que tous les mois ont 30 jours. À noter que les dates de naissance et de mariage sont données par les champs DN et DM.

Tips : si a_m, m_m, d_m sont resp. l'année, le mois et le jour de mariage et a_n, m_n, d_n de même pour la naissance, l'âge d'une personne lors du mariage peut être exprimé par la formule

$$A = (a_m + \frac{m_m}{12} + \frac{d_m}{360}) - (a_n + \frac{m_n}{12} + \frac{d_n}{360})$$

[ ]:
```
compteur=0
total=0.
r = re.compile(r"^[0-9]+;[^;]*;[^;]*;([0-9][0-9])/([0-9][0-9])/([0-9]{4});[^;]*;
 ↪([0-9][0-9])/([0-9][0-9])/([0-9]{4})")
f = open('Personnels.csv','r')
for ligne in f:
    for m in r.finditer(ligne):
        datenaissance=float(m.group(3))+((float(m.group(2))-1)/12)+((float(m.
 ↪group(1))-1)/360)
        datemariage=float(m.group(6))+((float(m.group(5))-1)/12)+((float(m.
 ↪group(4))-1)/360)
        print(datemariage - datenaissance)
        total = total + datemariage - datenaissance
        compteur = compteur+1
f.close()
m=float(total/float(compteur))
print(round(m,6))
```

11.5 EXO 5

Écrivez une expression régulière qui identifie :

- Nom de variable: Un identificateur est une suite de caractères qui n'est pas un nombre. Il s'agit d'une suite de caractères contenant des lettres non accentuées majuscules ou minuscules, des chiffres, et les symboles - ou underscore (_). Un identificateur commence toujours par un caractère différent d'un chiffre (c'est-à- dire une lettre majuscule ou minuscule). Un identificateur de fonction et de variable peut avoir un seul caractère.
- Nombre entier `15,000312`
- Nombre décimal `+.3,0.369,003.021,-003.021`
- Nombre hexadécimal: Un nombre hexadécimal est une séquence de chiffres et de caractères [A - F] se terminant par un h minuscule. Exemple : `AB4E51h,AAAh`
- Les commentaires : Les commentaires commence par les caractères (#) et se terminent par les caractères (#/) ou commencent par les caractères (#) se terminent à la fin de la ligne.

[ ]:
```
def Check_score(reg,Accepted,NotAccepted):
    r = re.compile(reg)
    print('Accepted strings are:')
    i=0
    score =0
    for m in Accepted:
        if bool(r.fullmatch(m)):
            i+=1
```

```
            print(f'This string is accepted : {m}')
    if i == len(Accepted): print(' All strings has been accepted')
    else: print('Your regular expression is wrong')
    score=len(Accepted)-i
    i=0
    print('---------------')
    print('Not Accepted strings are:')
    for m in NotAccepted:
        if bool(r.fullmatch(m)):
            i+=1
            print(f'erreur this is accepted : {m}')
        else:
            print(f'This string is not accepted : {m}')
    if i == 0: print(' All strings has not been accepted')
    score=score+i
    s=len(Accepted)+len(NotAccepted)
    print(f'your score {((s-score)/(s))*100}')
```

[ ]:
```
reg =r'\d*'
Accepted = ['12','2312','1231','1231']
NotAccepted=['*89','Co;)','_','-','-op','_op']
Check_score(reg,Accepted,NotAccepted)
```

[ ]:
```
reg =r'(^[a-zA-Z1-9][a-zA-Z1-9-_]+)'
Accepted = ['aa','a2','12a','A12_ds']
NotAccepted=['*89','Co;)','_','-','-op','_op']
Check_score(reg,Accepted,NotAccepted)
```

[ ]:
```
reg=r'^[-+]?\d*.?\d+'
Accepted = ['1','22','133','.3','-0.3','+.0365','-.0365']
NotAccepted = ['-','+','0.3,3','0..3','+-.8890']
Check_score(reg,Accepted,NotAccepted)
```

[ ]:
```
reg =r'[A-F\d]*\.h$'
Accepted = ['12h','AB12h','AACFh','1231']
NotAccepted=['*89','Co;)','_','-','-op','_op']
Check_score(reg,Accepted,NotAccepted)
```

[ ]:
```
reg =r'(\\#.*#/)|(\\#.*$[\r\n]+)'
Accepted = ['\#sdfsad#/','\# 515 1526 sdfsd 565#/
 ↪','\#sdfsdfs\r','\#sdfsdfs\n','\#sdfsdfs\r\n']
NotAccepted=['\# 515 1526/','\ 515 1526/','\#sdfsdfs\n\r\n']
Check_score(reg,Accepted,NotAccepted)
```

12 TP 11 : Cryptographie

12.1 Fonction de Hachage

Plusieurs algorithmes de hachage existent actuellement : MD5, SHA, CRC. La bibliothèque python standard inclut ces fonctions dans la bibliothèque hashlib. Ces algorithmes sont très similaires aux algorithmes de cryptographie AES et DES. Chaque algorithme produit une valeur de hachage de taille fixe quelle que soit la taille des données d'entrée. Par exemple, SHA256 produit une valeur de taille 256 bits (32 octets), SHA1 produit des valeurs de 160 bits (20 octets) et MD5 avec 128 bits. Les mots de passe des utilisateurs sont souvent enregistrés sous forme de hash au lieu de leurs valeurs claires. ci-dessous, un exemple d'utilisation des fonctions de hachage.

[1]:
```
# importe les fonctions sha256, sha1 et crc32
from hashlib import sha256
from hashlib import sha1
from zlib import crc32
# crée un message encode en octets (le b au debut # de la chaine de caracteres␣
 ↪signifie byte)
message=b"Un message a transporter"
# calcule et affiche le hashage du message
# avec chaque algorithme
print("sha256 = ",sha256(message).hexdigest().upper())
print("sha1 =", sha1(message).hexdigest().upper())
print("crc32 =",hex(crc32(message))[2::].upper())
```

```
sha256 =  16F71232D915F3533D6510E032EB1C4BFEF2AF39E3EB44A0B58B9C44D4A65361
sha1 = 2B3A855FBA7F4A19090DCB61AC5C7011E7D12AFC
crc32 = CB554B3E
```

Travail demandé - Écrivez un programme permettant de lire un fichier comme une suite d'octet (de préférence le fichier ne doit pas dépasser 1Mo). Concaténez les octets lus dans une variable appelée "data" - Calculez le hash SHA256 et MD5 de data. - Changez un seul caractère du message claire et comparez le résultat produit après modification avec le résultat du hashage avant modification. - Est-ce que la propriété de diffusion est assurée par les algorithmes de hashage? Est ce que deux messages / fichiers différent peuvent avoir le même résultat de hashage ?

12.2 Chiffrement avec AES

La bibliothèque Crypto offre l'implémentation de plusieurs algorithmes de chiffrement comme AES et RSA. Pour commencer, il faudra générer une clé primaire aléatoire selon la taille de la clé supportée par l'algorithme. Ceci peut être assuré par la fonction suivante cle_16_octet =os.urandom(16). Le code suivant montre comment chiffrer un contenu avec AES en utilisant la bibliothèque de cryptographie Crypto :

[2]:
```
import os
from Crypto.Cipher import AES

# os.urandom(N) genere une sequence
# de N octets aleatoire
```

```python
# cle de 16 octets (128 bits) aletoire
cle_16_octet =os.urandom(16)

#cree une instance AES avec une cle= "cle16 octets AES"
#la taille de la cle ==  (16octets)
objet_de_chiffrement=AES.new(cle_16_octet,  AES. MODE_ECB)

Message_claire=b"Message claire16"
print(Message_claire, len(Message_claire))

#chiffre le message claire
contenu_chiffre=objet_de_chiffrement.encrypt(Message_claire)
print(contenu_chiffre)

#Dechiffre le message chiffre
objet_de_chiffrement=AES.new(cle_16_octet,  AES. MODE_ECB)
Message_claire=objet_de_chiffrement.decrypt(contenu_chiffre)
print(Message_claire)
```

```
b'Message claire16' 16
b"\xd2^\x94)VO['\n\xf3\\\xed\xef\xeb5\x03"
b'Message claire16'
```

12.2.1 Important:

- Le texte à chiffré doit être d'une taille multiple de la taille de la clé. Sinon des octets de bourrage doivent être ajoutés à la fin du contenu.
- Pour indiquer le nombre de octets de bourrage à la fin du contenu, il existe plusieurs méthodes :

1. Ajout de la taille du message original sans bourrage avant le contenu chiffré.
2. jout du nombre d'octets de bourrage avant le contenu chiffré. -Utiliser un caractère absent du contenu claire pour les octets de bourrage pour marquer la fin lors du déchiffrement.

- Lors de l'initialisation de l'algorithme, AES automatiquement obtient la taille de la clé (128 bits, 192 bits ou 256 bits) pour déterminer la taille du bloque.

12.2.2 Travail demandé

Écrivez un programme qui chiffre le contenu d'un fichier et l'enregistre avec AES 128bits (16 octets) - Écrivez un programme permettant de lire le contenu d'un fichier sous forme de séquence d'octets. - Le programme doit calculer la taille en nombre d'octets du contenu. A partir de la taille, le programme doit calculer le nombre d'octets de bourrage à ajouter à la fin (entre 0 et 15). Cette valeur doit être sauvegardée dans une variable bourrage. - Ajoutez les octets de bourrage selon la valeur obtenue. - Générez une clé de 16 octets et sauvegarder la clé dans fichier key.bin afin de l'utiliser lors de la question de déchiffrement. - Chiffrez le contenu avec AES. - Ajoutez un octet au début du contenu chiffré contenant la valeur de bourrage. - Enregistrez le résultat dans un fichier.

12.3 Chiffrement avec RSA

12.3.1 Génération de cle RSA

RSA est un algorithme de chiffrement asymétrique. Celui-ci nessite la generation d'une cle publique et d'une cle privee. La cle publique est sauvegardée dans « fichier_cle_publique.pem » La cleprivée est sauvegardée dans «fichier_cle_prive.pem». Le programme suivant illustre comment génère des clés RSA.

[3]:
```
# importe le module RSA
from Crypto.PublicKey import RSA

# genere une cle publique/prive avec modulos (N) de 2048 bits
cle= RSA.generate(2048)

# recupere la cle prive
cle_prive = cle.exportKey()

#ouvre un fichier en mode d'ecriture en octets
Fichier = open("fichier_cle_prive.pem", "wb")

# ecrit le contenu de la cle prive sur fichier
Fichier.write(cle_prive)
# ferme le fichier
Fichier.close()

# recupere la cle publique
cle_publique = cle.publickey().exportKey()

# ecrit le contenu de la cle publique sur fichier
Fichier = open("fichier_cle_publique.pem", "wb")
Fichier.write(cle_publique)
Fichier.close()

#imprime la cle publique
print(cle_publique)

#imprime la cle RSA
print(cle_prive)
```

```
b'-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAsIZasA
GSCZUTkNV6Upeo\nSTnFQbPmQEwWCA2thzrMrSFUWnKfGCRbJo6vMlo46zWVzyVCFr1UBpnSOmuirbg6
\njgm07XSPoUk4ltSyUw57QxqhGrKjPOMsNnAllDv2c6+cczgjuzrAvMSQ6H8ur9zF\n0v61ei4XKstq
Y+ZPboyXGe2vITpw0tq7gIp+l44CtRE34xNfZ46G+z65ee3lKRGU\naQNXydxfkjXtBJUvWUme7hTLex
OEpTAtajBYk7Ao6ihUL5I2FrbObYG7Ts6/Tnz4\ntOfwYFdVlD8P/YIniaXi5UWQLI2WAJYJ4WLc2DUE
Wlet4vEDyWRrdzw4/C2wWnC3\nRwIDAQAB\n-----END PUBLIC KEY-----'
b'-----BEGIN RSA PRIVATE KEY-----\nMIIEogIBAAKCAQEAsIZasAGSCZUTkNV6UpeoSTnFQbPmQ
EwWCA2thzrMrSFUWnKf\nGCRbJo6vMlo46zWVzyVCFr1UBpnSOmuirbg6jgm07XSPoUk4ltSyUw57Qxq
hGrKj\nPOMsNnAllDv2c6+cczgjuzrAvMSQ6H8ur9zF0v61ei4XKstqY+ZPboyXGe2vITpw\n0tq7gIp
```

```
+l44CtRE34xNfZ46G+z65ee3lKRGUaQNXydxfkjXtBJUvWUme7hTLexOE\npTAtajBYk7Ao6ihUL5I2F
rbObYG7Ts6/Tnz4tOfwYFdVlD8P/YIniaXi5UWQLI2W\nAJYJ4WLc2DUEWlet4vEDyWRrdzw4/C2wWnC
3RwIDAQABAoIBAA5eWUpjTlZOVqHe\n5o6Lfqy7MZW4BYFtUAvGxT2xKRDcmiekldOavfqk+ZjBJEns+
i06zKbaMVk6MxUD\nRuIaEVYXby025CwOMDeCx6MgFiNvqvZn3QarPSGPPlfTC6v9+t+cDbiDP0X+QAd
U\ntuGkuEuE5RO1Zbhbh2TrTcLjWbdH8VFNJvWufxqSbROt1gxslotwaHWqApivyvnK\n0JISQx8MAbm
HKGLS6ADR9yohBaJWjURjgW0CmkJKHNO8yao15zrRo1SgjdqJB3EB\n+p6PfUaI1A09LGA1KbXe/DNvH
tqkxmBonl1KdQmhe/LKC99WeeVcm5BhQbsTpR/S\n4Iuyfe0CgYEAwvhZpoMd6t9V/+cgxP7eGWzxv/x
TZtFyYP1B4e5QKf9wnjJf/KEf\noyinuEbBDYyk7aL8rl3LAfbKszq2fcbEYf556miVtasAaUAhtotj1
JsukgGaeEUU\nrwAK23lwzUP4gjhcR/h7SR9K+GXrMTmq6y5y+fnCcA6URFPH6HrC570CgYEA58fq\ns
AVti+3wV7d2G7LbnJOZYqkjjN/Bd944fAZJewzHgBlBUMdSecQHw5xyQ57OfzAK\ndvJtrMmj7kA1CUU
NQYH8OICd5PNpfq+QhbmRK66CtFcMcxcWg+B4kL17Zjc87dcQ\nLc+hIb2hFy6lvahroKoN8Du2tU3cI
nccvztouVMCgYAycADxPJYuvpwG2Yn2rGBU\nf4SCwAnrXV+Ti7DRe88tLjG6GxoNxrjigo/w8gzbLnk
OKKfpi8YKugdyGkw/eX4w\nQ57Sbz/bgWNX1wlhqemnhIWlq9iEKIrTQtWMNXxi/aR6O45T6AosvnWsk
OEpGNwI\nfzmrRVtxlbPxgSMEF034mQKBgE7jamuSzWBNEfqpBNgUnk7CpfhAnUr7dXv49Lyx\n3xy0k
sp0zhhSXy32nLXlb7a4PIxpq4YkqVOgJd2XmPWYvRRIy+WjgRpGBugJlWUO\nbeyyLuZ2olQhzZYBNsf
81h1KO8Cv7F/LRYqYYlXjmKHeyxMaDjhv+pXm9D/+zpf/\n/nrlAoGANlh7uiawCiHxJDr/TD9OGjZh0
Ju1tu3tTZJE+nhP+zqPII8zSopwuxiD\nvwb53HpcgOO+6lru0OO33LRk7zcb3QQ5p6QQzlTXR9lr7Ul
5TpW9pQOVodaFIxDx\nscrrP+fuzmWapxMlLAki5JG/hpp9mEmtgefixmRc+xKt0AOXSJQ=\n-----
END RSA PRIVATE KEY-----'
```

12.3.2 Chiffrement et déchiffrement avec RSA

Le programme suivant permet d'importer les clés publique et privee ainsi que le chiffrement avec cle publique et dechiffrement avec cle privee. Un objet permettant de chiffrer avec la cle publique est affecte aa objet_cle_rsa. Un objet permettant de chiffrer avec la cle privee est affecte a objet_cle_rsa_prive.

[4]:
```
import Crypto
from Crypto.PublicKey import RSA
from Crypto.Cipher import  PKCS1_OAEP

# importe la cle publique du fichier
contenu_clepublique = RSA.importKey(open("fichier_cle_publique.pem","rb").read())

# cree un objet a partir de la cle permetant de chiffre avec RSA
objet_cle_rsa = PKCS1_OAEP.new(contenu_clepublique)

# message claire
message=b"Message"
# chiffre le message avec la cle publique
message_chiffre=objet_cle_rsa.encrypt(message)

#imprime le resultat du chiffrement
print(message_chiffre)

# importe la cle prive du fichier
contenu_cleprive = RSA.importKey(open("fichier_cle_prive.pem","rb").read())
# cree un objet a partir de la cle permetant de chiffre avec RSA
```

```
objet_cle_rsa_prive = PKCS1_OAEP.new(contenu_cleprive)
#imprime le resultat du dechiffrement
print(objet_cle_rsa_prive.decrypt(message_chiffre))
```

```
b"\x01\xac;\xf8l:\xb1\xac/\xc7\xc5\xfd\xf9\x82\xe6\xc6mB\xc5\xde\xd2L~\x10_\x9f\
x95J\xc3%\x12pSl4\xd3\xf5\x19}\x89\xb8\xb4\xb0\xd3]%A\x0f\xab\x1e\xca\xae\xb4\xc
b\n\x11\x9d\xa0\xb0\x1c\xf8\x82\xf3C\xc0\x93ah!\tb!g\xb4x\x84\x87\x85\x14s)Sr\xe
7\xbe\x93y\xd4\x80\xc1\xe1p\xaa\x19E\x9a\xa7\x8f\x86\xd6\xee\xc8<:\x17=\xe1\x83\
xd2\x8b\xc1\x03\x05\x17\xf3\xf8\xa0\xfa\xd0\x00\xb8\xb7\xc4\x11HL\xfe-\xdf\x030\
x9a+i\xce\x92\xd4{\x96\x15f7\xee\xcc\xeb\rS\x0f\r\x1b\x86\xdc\x1bK\\\xac\x8f\xe1
\xa9\x9d\xb2\x88\xc5A\x84\xa2&\x7f\xf7\xe8S-.\xb1\xf1l\xba\xcbD\xa5 \x1eE.y\x1a\
x08\n\x03\x87\xc1\xe2m\x85\x94n\x1dt\xe0\xbb\x0b\xc4\xfc\xde\xebx\x0f\x11\x93S1\
xc4\xbc<Bz'\xd8`\x7f\xfa\xd0\x919\xbb\xd6H\x1e\xaa\x9b\xf4\x94\xe4\x18\x82v\x1a\
x17vW\x96\xb37eq\x93:\x1b\x8e!\xea*\xc7q~\xeb"
b'Message'
```