# DOSSIER DE CONCEPTION

TP n°2 : Sujet n°4
Implémentation de concepts orientés objet en Python & Analyse de données

**Nathan AMSELLEM**
**Valentin THEDON**

03/11/2022

—

Cours de Programmation Orientée Objet Avancée

—

Professeur : Hamid MCheick

# Table des matières

# Question n°4

## Principes de conceptions

Dans ce TP il nous est demandé d'introduire des concepts de programmation objets en python et de montrer une étude de cas de ces concepts :

- Polymorphisme
- Surcharge des méthodes (overloading),
- Redéfinition des méthodes (overriding)
- Généricité
- Modularité

Nous tenterons donc à travers une étude de cas que nous avons imaginé de présenter la manière d'introduire ces concepts de programmation.

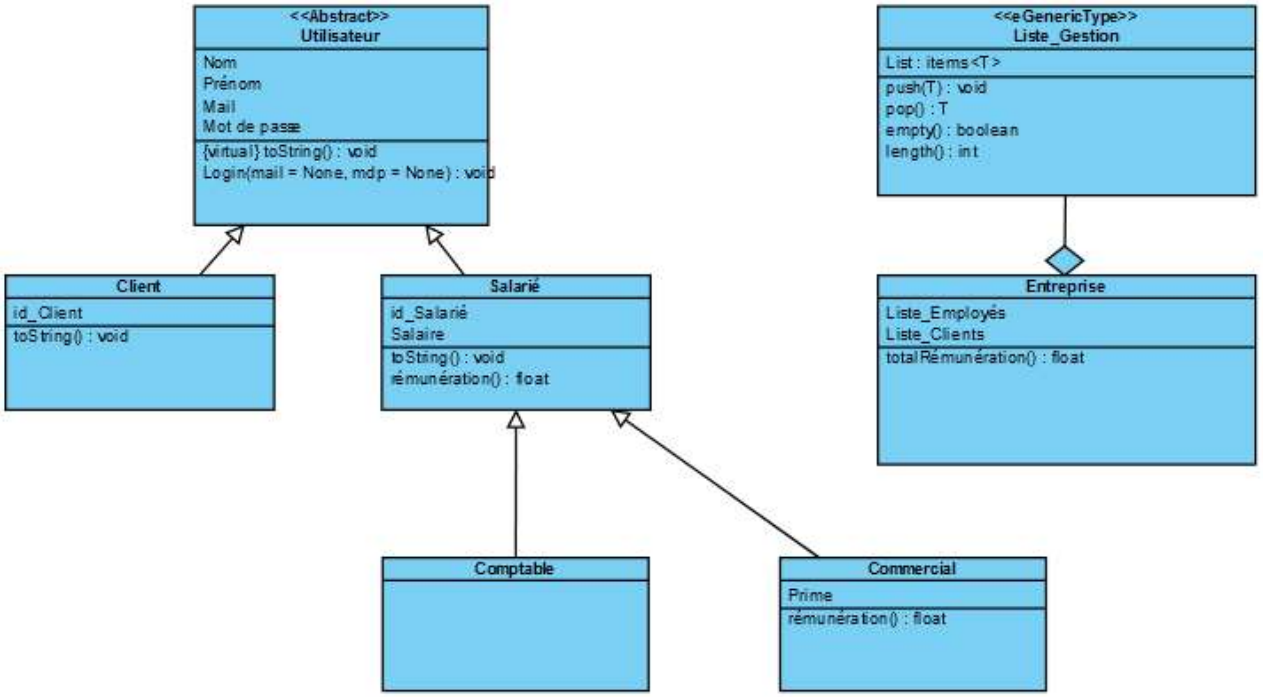
## Cas de figure

Soit un système de gestion d'entreprise simplifié qui comporte un certain nombre de clients et d'employés. On considèrera que cette entreprise est composée d'employés commerciaux et comptables.

Le système devra être capable de créer des clients, des employés (qu'ils soient comptables, commerciaux ou ayant un statut indéterminé) et de les ajouter à l'entreprise à travers des listes qui seront des attributs de cette classe.

Une fonction toString() sera implémentée afin de connaitre le statut de l'instance créée, de plus l'entreprise devra être capable de calculer la somme des rémunérations de tous les employés de l'entreprise. Le système sera aussi capable de vérifier les identifiants de connexion rentrés par un utilisateur.

# Modélisation Objet




A travers cette modélisation nous pourrons mettre en œuvre les concepts clés énoncés précédemment avec notamment :

- Le Polymorphisme : qui fait référence à la capacité d'une variable, d'une fonction ou d'un objet à prendre plusieurs formes, c'est-à-dire à sa capacité de posséder plusieurs définitions différentes
    - La fonction toString() d'équivalent « virutal » pour du C++ par exemple ne renverra aucune action : l'utilisateur dérivera donc de la classe abc qui définit des classe abstraites
    - toString() sera donc obligatoirement définie dans les classes Client et Salarié

- L'overriding qui permettra de redéfinir d'une façon différente une fonction d'une classe fille présente dans une classe mère
    - Il sera illustré à travers la redéfinition de la fonction rémunération () :
        - Rémunération() est définie dans la classe Salarié : cette fonction renverra le salaire mensuel d'une personne
        - Rémunération() est redéfinie dans la classe Commercial qui hérite elle-même de Salarié : cette fonction renverra le salaire

mensuel d'une personne en prenant en compte les primes mensuelles du commercial

- L'overloading permettra quant à lui de redéfinir une même méthode dans une même classe avec des paramètres différents :
    - Il sera illustré à travers la fonction Login() qui pourra être définie avec les paramètres :
        - Mail : adresse mail de l'utilisateur
        - Mdp : mot de passe de l'utilisateur
    - Si la méthode reçoit 2 paramètres corrects alors il sera connecté
    - Si la méthode 2 mauvais paramètres alors il ne sera pas connecté
    - Si la méthode reçoit 1 paramètre ou aucun des deux alors il ne sera pas connecté

- La Généricité consiste à définir des algorithmes identiques opérant sur des données de types différents. On définit de cette façon des procédures ou des types entiers génériques :
    - Il sera illustré à travers la classe Liste_Gestion qui fera dériver des types génériques T.
        - A travers cette classe on pourra donc gérer des listes d'un Type T à instancier et ainsi surcharger les fonctions qui sont utiles à la gestion que l'on prévoit d'effectuer sur nos employés.
    - La généricité sera ensuite utilisée dans la classe Entreprise puisque ses attributs seront à la fois des listes de clients et des listes d'employés.
        - On cherchera donc à uniformiser la gestion de ses deux attributs

- La modularité :
    - On créera dans notre cas d'étude un module par classe présentée dans le diagramme plus haut à savoir :
        - Entreprise.py
        - Liste.py
        - Type_Salarie.py
        - Salaries.py
        - Clients.py
        - Utilisateurs.py

# Analyse de l'implémentation

Après avoir commenté les différents aspects conceptuels et dans quelle partie du programme ces concepts trouvent leur place dans le code, nous allons analyser la manière dont ils sont implémentés et les avantages/inconvénients qu'ils présentent par rapport à des langages objets classiques du type C++.

## Polymorphisme

**toString():**

Une méthode est dite **virtuelle** si, lorsqu'elle est redéfinie dans une classe dérivée, le type d'instance (classe fille ou mère) détermine la méthode réelle à appeler lors de l'exécution (permet le **polymorphisme**)

Dans l'implémentation présente nous souhaitons que **toString()** s'adapte au statut de l'utilisateur de l'entreprise : Client ou Employé

Nous définissons donc cette fonction dans la classe utilisateur :
Ici, nous utilisons un nouveau mot clef **pass** qui sert à créer une fonction vide. En effet, le mot clef **pass** ne fait strictement rien en Python. On est obligés de l'utiliser pour créer une fonction vide car si rien n'est écrit dans notre fonction l'interpréteur Python va renvoyer une erreur.
Ma classe utilisateur disposera donc d'une fonction toString qui ne contient pas d'instructions.

Dans la mesure où cette fonction ne contient pas de d'instructions lui étant propre il conviendra d'y rajouter le décorateur **ABC** de la metaclass **abc** qui définira le fait qu'aucune classe **Utilisateur** ne pourra être instanciée de plus, aucune des classes dérivées ne pourront l'être tant que la méthode **toString()** ne sera pas redéfinie dans les classes dérivant de **Utilisateur** : à savoir **Client** et **Employé**

## Overriding

**Remuneration() :**

L'overriding se présente de la même manière que dans les programmes orientés objet classiques.
La classe parent et enfant devront avoir la même signature. Ils devront avoir le même nombre de paramètres. De plus, le type renvoyé pourra différer entre les 2 définitions mère/fille. Il permettra à la classe fille d'adapter grandement l'exécution de toute méthode définie dans la classe mère.
En revanche il faut préciser que la différence majeure de python est qu'il est un langage interprété et non compilé. L'interpréteur ira donc chercher la définition la plus proche : c'est-à-dire celle l'instance de classe fille en question.

NB : Pas d'overriding admit pour les fonctions du type **static/private/final.**

## Overloading

**Login():**

La surcharge de méthodes ou overloading est un sous partie du polymorphisme. Il représente le procédé par lequel un même « titre » de fonction peut être redéfini plusieurs fois dans une même classe.

Python ne supporte pas l'overloading : plusieurs redéfinitions d'une même fonction différent selon les arguments passés en en-tête.

Python renverra une erreur à l'interprétation si on effectue une implémentation du type C++/JAVA. Ceci s'explique par le fait que python ne soit pas un langage typé ou plutôt « auto-typé ».

Comme j'ai pu le vérifier d'après la source suivante :
https://www.scaler.com/topics/function-overloading-in-python/

```
class sumClass:
    def sum(self, a, b):
        print("First method:",a+b)
    def sum(self, a, b, c):
        print("Second method:", a + b + c)

obj=sumClass()
obj.sum(19, 8, 77) #correct output
obj.sum(18, 20) #throws error
```

Cette implémentation renverra une erreur car python prendra seulement en compte la dernière définition de cette méthode : c-a-d la définition à 3 paramètres.

En revanche il est possible d'implémenter cette caractéristique dans un programme Orienté Objet en python :

Il suffira d'envoyer **None** sur un attribut ou plus passés en argument de notre méthode. La manière dont procèdera cette méthode diffèrera selon les cas en regardants quels attributs sont envoyés en argument.

NB : Les méthodes du type **static/private/final** pourront être 'overloadées'

## Généricité

**Liste_Gestion :**

En python la généricité trouve son sens au travers de l'import du module typing intégré à python et de ses générateur de typages TypeVar et Generic.

TypeVar permettra de déclarer un type de variable, ils servent de paramètres pour les classes et fonctions génériques.

Comme évoqué plus haut : le générateur Generic nous permettra de définir des fonctions génériques :

```
T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

Ainsi que des classes génériques :

```
T = TypeVar('T')

class LoggedVar(Generic[T]):
```

Dans le cas des classes génériques (implémentées dans notre code) le type 'T' sera donc rendu valide en tant que paramètre ainsi qu'en type de retour et dans le corps de nos méthodes : le type est reconnu.
Source : https://docs.python.org/3/library/typing.html

La classe Liste_Gestion nous permettra de gérer des listes d'un type souhaité : dans le cas présent des listes de clients et d'employés.

### Modularité

En python il y a plusieurs méthodes d'import de modules :

```
from Utilisateur import Utilisateur
from Clients import Client
from Salaries import Salarie
from Types_Salaries import Comptable, Commercial
```

Les imports ci-dessous signifient que l'on veut utiliser la classe client et toutes ses méthodes implémentées dans « Clients.py ». Même chose pour les salariés.

```
from Types_Salaries import *
```

Cet import prend en compte toutes les fonctions et classes du module grâce à l'argument « * ».

Pour éviter des confusions entre deux fonctions portant éventuellement le même nom dans deux modules différents, il peut être utile d'importer un module en lui fixant un nom d'utilisation en ajoutant en début de programme :

```
import numpy as np
```

Auquel cas chaque fonction de la bibliothèque numpy sera appelée avec le préfixe « np. ».

## Analyse de données

Dans ce TP il nous est aussi demandé de montrer les avantages de Python dans les analyses de données avec une étude de cas.

Nous tenterons donc à travers une étude de cas que nous avons imaginé de présenter la manière d'introduire ces concepts de programmation.

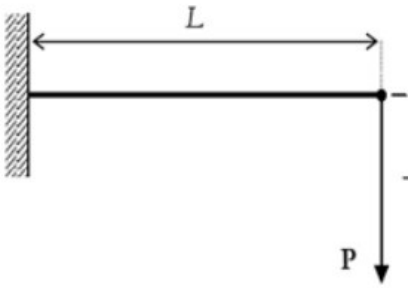
# Cas de figure

Dans la mesure ou l'un d'entre nous a actuellement des cours d'apprentissage automatique avec des activités d'analyse de données, il nous a semblé pertinent de reprendre ce cas de figure en essayant d'édulcorer au maximum les dépendances au problème tout en montrant les concepts intéressants autour de la programmation python en analyse.

Ceci a aussi permis de rendre le code de génération de données (auparavant procédural) un peu plus OO pour rester dans le cadre du cours. La création de certaines classes rend forcément le code plus structuré donc plus lisible.

Le problème dans lequel on se place est un problème de résolution par **éléments finis :** la méthode des éléments finis est utilisée pour résoudre numériquement des équations aux dérivées partielles.
Cette méthode permet notamment de discrétiser numériquement des structures physiques et d'extraire des données d'intérêt.

On se placera physiquement dans le cas d'une poutre encastrée en l'une de ses extrémités et libre sur l'autre avec un poids P appliquée sur cette extrémité dont voici le schéma :




Cette poutre réagira au poids appliqué en résonnant à certaines fréquences (en réalité une infinité).

Notre but sera donc de modéliser un dataset d'entrées sous forme de csv et d'en extraire un autre dataset de sortie sous forme csv au travers d'une résolution par éléments finis sur une poutre encastrée libre. Nous effectuerons ensuite une analyse de données sur ces 2 csv.

## Entrées

Les entrées du problème seront les caractéristiques physiques des différentes instances de poutres générées avec 10 entrées :

- NbElts
- L_tot
- Rho
- H
- B
- S
- I
- L
- E
- Mat

Ces caractéristiques résultent de la modélisation physique du code 'generate_data_learn_V1_10elts.py'.

Au niveau de l'analyse de données nous nous intéresserons notamment aux matériaux 'Mat' et notamment des matériaux en 'Aluminium'.

## Sorties

Les sorties du problème seront les caractéristiques de fréquence des différentes instances de poutres générées avec 8 fréquences dites de résonnance :

- Freq1
- Freq2
- Freq3
- Freq4
- Freq5
- Freq6
- Freq7
- Freq8

Ces caractéristiques résultent de la modélisation physique du code 'generate_data_learn_V1_10elts.py'.

Au niveau de l'analyse de données nous nous intéresserons aux fréquences 'Freq1' et notamment des instances dont les matériaux correspondent à de l'aluminium.

## Objectif de l'analyse de données

A travers cette analyse de données nous explorerons :

- Les avantages et les facilités du langage python vis-à-vis de l'analyse de données
- Les avantages et les facilités offertes par les librairies développées par la communauté vis-à-vis de l'analyse de données

# Analyse de l'implémentation

## Pandas & Seaborn

### Exemples d'implémentation

En analyse de données on utilise très souvent l'extension .csv afin de lire des tableaux de données imposants.
Le premier objectif sera donc d'arriver à lire un tel fichier à travers des librairies spécialisées afin de convertir le fichier en une instance interprétable dans le code.

La démarche sera donc la suivante :

1. A travers la fameuse librairie **PANDAS** de lire le fichier « dict.csv » ainsi que « test.csv » afin de les convertir en une structure de données bidimensionnelles (alignées de façon tabulaire en lignes et colonnes) appelée **DataFrame**

2. Examiner le **DataFrame** et la façon dont nous pouvons manipuler à la fois le **DataFrame** et les données qu'il représente pour construire une base pour réaliser des données interactives.

3. Utiliser **Seaborn**, une librairie de data visualisation basée sur la librairie **Matplotlib** offrant une présentation des données interactives et simples à mettre en place sur les **DataFrames** en question.

Implémentation :

1. Nous chargeons les 2 tableaux de données dans un DataFrame que nous pouvons utiliser pour voir diverses opérations :

```
self.freq = pd.read_csv(open(path_freq, "r"),
                        delimiter=",")
self.inputs = pd.read_csv(open(path_inputs, "r"),
                          delimiter=",")
```

On peut éventuellement examiner les cinq premières lignes du DataFrame en utilisant la méthode **freq.head()** ou **inputs.head()** afin d'observer le dataframe en limitant le nombre de lignes d'affichages.

| | freq1 | freq2 | freq3 | ... | freq6 | freq7 | freq8 |
|---|---|---|---|---|---|---|---|
| 0 | 102.243401 | 640.769287 | 1794.570068 | ... | 8728.655273 | 12246.726562 | 16407.236328 |
| 1 | 89.510002 | 560.967590 | 1571.073486 | ... | 7641.584473 | 10721.513672 | 14363.873047 |
| 2 | 60.865200 | 381.447906 | 1068.301758 | ... | 5196.139648 | 7290.436035 | 9767.174805 |
| 3 | 33.677502 | 211.059998 | 591.104980 | ... | 2875.090576 | 4033.891602 | 5404.302734 |
| 4 | 13.668600 | 85.662300 | 239.910095 | ... | 1166.905029 | 1637.224243 | 2193.429199 |

Ceci nous permet de confirmer que le dataFrame se génère correctement.

Admettons maintenant qu'en vue de l'étude qui sera menée nous voulons préalablement analyser les poutres générées avec comme matériau de l' « Aluminium ».
Nous extrayons alors les entrées dont le label est aluminium pour l'attribut matériau « Mat ».
Pour cela j'ai choisi de créer une méthode appartenant à la classe **DF_Cantilever_beams.** En réalité, les analyses qui sont menées couramment dans les cours d'intelligence artificielles ne prennent pas la peine d'utiliser des classes, la programmation est souvent purement procédurale.

Afin de localiser ces poutres nous utilisons la fonction **.loc[]** sur notre dataframe :

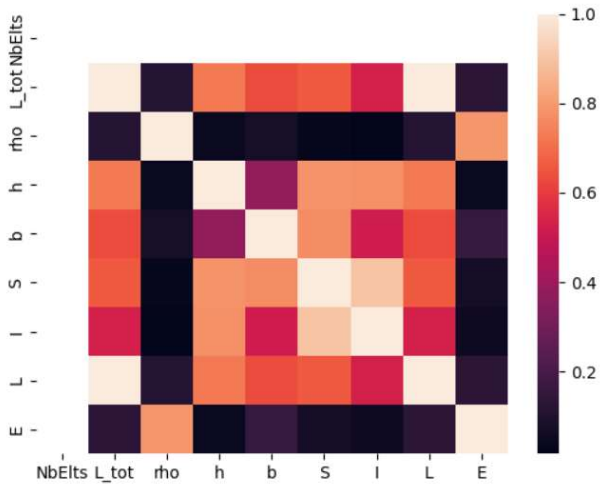
```
Material = self.inputs.loc[self.inputs.loc[:, 'Mat'] == mat]
```

Le DataFrame de sortie étant trié de la même manière, nous pouvons donc de la même manière récupérer en 2 ligne un DataFrame entier au travers des indexs récupérés en entrée :

```
indexs = self.isolate_material_inputs(mat).index
freq_alu = self.freq.iloc[indexs]
```

Nous pouvons alors mener notre analyse sur ces données…

Supposons qu'au cours de l'étude nous nous apercevons que certaines entrées sont correlées, elles ne sont donc pas forcément nécessaires dans le cas d'un apprentissage automatique dans la mesure où l'information contenue dans une variable sera corrélée à l'autre. Pour visualiser cela, nous allons pouvoir utiliser la bibliothèque **Seaborn** et sa **heatmap** qui montre la **matrice de corrélation** de toutes les variables d'entrées.




Grace à un seul appel dans la librairie **Seaborn** on se retrouve très facilement une représentation d'un outil mathématique très puissant que l'on peut visualiser puis analyser.
Elle permettra surement de rendre notre algorithme d'apprentissage automatique plus efficace avec la réduction de dimensionnalité qui découle de cette analyse :

```
# d'apres la matrice de correlation, certaines entrées sont étroitement liées
# on va donc supprimer certaines de ces valeurs pour conserver :
# surface S la masse volumique rho, la longueur de la poutre L_tot
corr = ['NbElts', 'rho', 'h', 'b', 'I', 'L']

#On supprime les labels contenus dans corr
Beams.reduce_inputs(corr)
```

## Avantages et caractéristiques de ces librairies & outils

**Pandas :**

- Une des bibliothèques Python les plus utilisées pour la Data Science.
- Développé en 2008 par Wes McKinney
- Open source
- Implémenté à partir de C – d'où sa rapidité
- A introduit les objets **DataFrame** et **Series**

Pandas peut être installé via pip depuis PyPI :

*$ pip install pandas*

**DataFrame**

- La structure de données la plus utilisée dans un projet de Data Science.
- C'est un tableau avec des lignes et des colonnes, comme une table SQL ou une feuille de calcul Excel.
- La table peut stocker en mémoire des données dans des formats différents.
- Offre une analyse de séries chronologiques très performantes.
- Un DataFrame contient une ou plusieurs Series.
- Un DataFrame est mutable.

**Series**

- Un objet Series est de 1 dimension comme un tableau. L'objet Series est un objet de données mutable. Cela signifie qu'il peut être mis à jour, que de nouveaux éléments peuvent être ajoutés et que des éléments existants peuvent être supprimés.
- Il peut contenir des données de tout type.
- Il peut être instancié avec un **tableau** ou un **dictionnaire**. Les clés du dictionnaire sont utilisées pour représenter les index.

**Seaborn** :

- Une bibliothèque qui offre la possibilité de résumer et de visualiser des données.
- Cette bibliothèque apporte des fonctionnalités inédites qui favorisent **l'exploration et la compréhension des données**.
- Son interface utilise **Matplotlib** avec de nombreuses fonctions intuitives qui assurent notamment la cartographie sémantique et aident à la conversion des données en graphiques statistiques à visualiser.

## Numpy & Listes

### Exemples d'implémentation

A partir de l'étude menée précédemment, nous allons vouloir montrer l'implémentation que l'on peut avoir de ces fonctionnalités grâce à la librairie **numpy** ainsi qu'au **slicing** sur des **listes** natives à python.

On commencera par la création de la classe Poutre encastrée libre en traitant des listes : types natifs à python afin de voir les possibilités qui sont offertes par de tels types

```
self.list_freq = list(csv.reader(open(path_freq, "r"),
                                 delimiter=","))
self.list_inputs = list(csv.reader(open(path_inputs, "r"),
                                   delimiter=","))
```

Nous chargeons les 2 tableaux de données dans une liste. Les listes sont natives au langage python.
Les listes Python sont par défaut indexées ou indicées. Cela signifie que chaque valeur d'une liste est liée à un indice qu'on va pouvoir utiliser pour récupérer cette valeur en particulier. Ceci est le **slicing**, un slice permet le découpage de structures de données séquentielles, typiquement les **chaînes de caractères** ou les **listes**.
Les slices sont des expressions du langage Python qui vous permettent *en* une ligne de code d'extraire des éléments d'une **liste** ou d'une **chaîne**. En voici un exemple :

```
# On isole les labels de données d'entrées et de sorties (freq)
# Slicing horizontal pour des listes 2d : natif à python
# "...[0]..." : selectionner la premiere ligne
# "...[:]" : selectionner toutes les colonnes
self.labels_inputs = self.list_inputs[0][:]
self.labels_freq = self.list_freq[0][:]
```

On pourra à la suite de cela convertir notre **liste** en **array** : structure de données native à la librairie **Numpy**. On récupèrera une variable particulière au travers du **Slicing Numpy**.

```
#slicing offert par la librairie numpy :
#   "...[: ..." : sélectionne toutes les lignes...
#   "... 0]..." : ... de la premiere colonne
arr_freq1 = arr_freq[:, 0]
```

## Avantages des librairies et du langage

Une liste est une structure de données intégrée à Python qui contient une collection d'objets. Les **listes** présentent un certain nombre de caractéristiques importantes :

- Les éléments de la liste sont placés entre **crochets**
- Les listes sont **ordonnées**, on pourra utiliser un index pour accéder à n'importe quel élément.
- Les listes sont **modifiables**
- La duplication des éléments est possible car chaque élément a sa propre place distincte accessible par **l'index**.
- Les éléments peuvent être de types de données différents : on pourra combiner des **chaînes de caractères**, des **entiers** et des **objets** dans la même liste.

On aura globalement les mêmes fonctionnalités pour un tableau Numpy avec quelques facilités d'accès aux différents éléments. On notera toutefois les différences suivantes :

- Les tableaux doivent être déclarés. Les listes ne le sont pas, puisqu'elles sont intégrées à Python.
- La création d'un tableau, en revanche, nécessite une fonction spécifique du paquet NumPy (c'est-à-dire **numpy.array()**). De ce fait, les listes sont statistiquement **plus utilisées** que les tableaux.
- Les tableaux peuvent stocker des données de manière très compacte et sont plus efficaces pour stocker de grandes quantités de données : caractéristique très importante et recherchée en analyse de données !
- Les tableaux gèrent mieux les opérations numériques que les listes.

# Conclusion

Python est l'un des langages de programmation orientés objet les plus populaires au monde. Il est très simple à appréhender.
Il présentera néanmoins certaines particularités quant aux pratiques classiques de l'OO avec par exemple les méthodes spéciales (__init__ pour un constructeur), l'impossibilité de créer un réel overloading…
Il est très utilisé en Data science. L'un des principaux avantages de Python réside surtout dans le nombre de bibliothèques externes qu'il possède. Celles-ci facilitent la vie aux programmeurs et rendent assez aisée la manipulation des données.