

# Projet Python

Le projet Python constitue l'examen final du cours *Python avancé*.

## Modalités

- Chaque groupe doit être constitué de **3 à 4 personnes** maximum.
- Une **présentation orale de 15 minutes** (incluant les questions) aura lieu lors du dernier cours. Lors de cette présentation, **tous les membres du groupe doivent prendre la parole** et présenter leur partie.
- Le projet doit être **fonctionnel, opérationnel et documenté** le jour de la présentation.
- Les livrables (présentation + code source) sont à fournir sur la plateforme MyGES. Un lien GitHub peut être envoyé ou supplément, mais **ne se substitue pas** aux livrables du code source du MyGES.

Pensez à **ne pas ajouter le dossier d'environnement virtuel** ( `env` ou `venv` ) pour éviter d'avoir un dossier compressé de plusieurs centaines de Mo !


## Description du projet

Le projet consiste à créer **MyBear**, une librairie Python de gestion de données alternative à Pandas. Pour cela, MyBear reprend le concept de DataFrame sous Pandas en y intégrant les fonctionnalités élémentaires.

## Notation

La notation tiendra compte des aspects suivants :

- (14 points) Code de la librairie.
  - (10 points) Implémentation de l'architecture de code et des fonctions demandées.
  - (2 points) Structuration du projet pertinente (architecture, PEP8, linting, refactoring, documentation type Docstring).
  - (2 points) Tests unitaires de l'API.
- (6 points) Présentation orale.
  - (2 point) Exécution valide en direct sur des exemples.
  - (1 point) Respect du temps imparti et répartition du temps de parole.
  - (3 points) Pertinence de l'exposé oral.

Tous les titres commençant par  indiquent les fonctions qui doivent être implémentées.

## Architecture

La librairie utilise deux classes : `Series` et `DataFrame`. Toutes les opérations sont effectuées à partir de ces deux classes.

Étant donné que le projet s'inspire de Pandas, il ne faut pas hésiter à regarder ce que sont les `Series` et `DataFrames` de Pandas.

### Classe `Series`

Une `Series` peut être vue comme une colonne dans un `DataFrame` qui contient en plus des données, une étiquette (un nom), et des informations statistiques déjà présentes et calculées automatiquement lors de la création de la `Serie` (taille, nombre de valeurs manquantes et type de données).

#### Implémentation de la classe `Series`

Implémenter la classe `Series` dont le constructeur prend en paramètre une liste de valeurs comme données, et un nom sous forme de chaîne de caractères.

#### Propriété `iloc`

Implémenter la propriété `iloc` permettant d'obtenir une indexation basée sur la position des éléments.

- Si `iloc[n]`, alors on retournera une seule valeur.
- Si `iloc[a:b]`, alors on retournera une `Series` qui contient les valeurs mentionnées.

```
In [ ]: @property
def iloc(self) -> Any | Series
```

#### Fonctions statistiques `max`, `min`, `mean`, `std` et `count`

Implémenter les fonctions statistiques précédentes, qui retournent pour chacune le calcul associé.

### Classe `DataFrame`

Un `DataFrame` contient un ensemble de `Serie` **ayant toutes les mêmes listes d'index**.

Les méthodes du `DataFrame` ne doivent jamais modifier l'objet courant : elles doivent tout le temps retourner un nouveau `DataFrame` résultant de l'opération effectuée.

#### Implémentation de la classe `DataFrame`

Implémenter la classe `DataFrame` avec plusieurs surcharges de constructeurs.

- Une version où l'on peut charger une liste de `Series` en tant que `DataFrame`.
- Une version où l'on peut directement les colonnes dans un paramètre, et les listes de valeurs dans un second paramètre (exemple, si l'on a 3 colonnes, alors il devrait y avoir une liste de taille 3 x *n* avec *n* le nombre de lignes).

#### Propriété `iloc`

Implémenter la propriété `iloc` permettant d'obtenir une indexation basée sur la position des éléments.

- Si `iloc[n, n]`, alors on retournera une seule valeur.
- Si `iloc[a:b, n]`, alors on retournera une `Series` qui contient les valeurs mentionnées.
- Si `iloc[n, a:b]`, alors on retournera un `DataFrame` qui contient une seule ligne et toutes les colonnes mentionnées.
- Si `iloc[x:y, a:b]`, alors on retournera un `DataFrame` qui contient les lignes et les colonnes mentionnées.

```
In [ ]: @property
def iloc(self) -> Any | Series | DataFrame
```

#### Fonctions statistiques `max`, `min`, `mean`, `std` et `count`

Implémenter les fonctions statistiques précédentes, qui retournent pour chacune le calcul associé à chaque colonne du `DataFrame`. Chaque fonction retourne un `DataFrame` avec les mêmes colonnes, mais avec une seule ligne correspondant au résultat de la appliquée sur chaque colonne.

#### Chargement de fichiers CSV ( `read_csv` )

La fonction `read_csv` est un membre statique de la classe `DataFrame` et permet de charger un fichier CSV en tant que `DataFrame`.

```
In [ ]: def read_csv(
path: str,
delimiter: str = ",",
) -> DataFrame
```

#### Chargement de fichiers JSON ( `read_json` )

La fonction `read_json` est un membre statique de la classe `DataFrame` et permet de charger un fichier JSON en tant que `DataFrame`. Les formats de fichiers (paramètre `orient`) possibles sont les suivants :

- `records` : [{column: value, ...}, ...].
- `columns` : {column1: [...], column2: [...], ...}.

```
In [ ]: def read_json(
path: str,
orient: str = "records"
) -> DataFrame
```

#### Groupement ( `groupby` )

Le groupement consiste à combiner et à agréger plusieurs lignes d'un `DataFrame` en formant des groupes à partir d'une ou plusieurs colonnes.

La fonction va prendre en paramètres `by` qui s'attend à recevoir le nom de la ou les colonnes sur lesquelles grouper, et `agg` qui va définir la stratégie d'agrégation sur les autres colonnes.

```
In [ ]: def groupby(
self,
by: List[str] | str,
agg: Dict[str, Callable[List[Any], Any]]
) -> DataFrame
```

Dans notre cas, toutes les colonnes autres que celles mentionnées par `by` doivent être définies dans le paramètre `agg`.

Le paramètre `agg` est un dictionnaire où chaque clé indique le nom de la colonne, et chaque valeur est une fonction qui va agréger une liste de valeur en une seule. On pourra tester par exemple avec des fonctions NumPy du style `np.max` ou `np.mean`.

Si l'on a par exemple 4 colonnes `a`, `b`, `c` et `d`, et que l'on réalise un groupement sur les deux premières colonnes, alors on aurait :

```
In [ ]: mon_dataframe.groupby(
by=["a", "b"],
agg={
    "c": np.min,
    "d": np.max
})
```

#### Jointure ( `join` )

La jointure est une opération qui permet de combiner des données provenant de deux `DataFrames` : celui depuis lequel la méthode `join` est appelée (*left*) et un autre `DataFrame` (*right*).

Le paramètre `left_on` permet de choisir la ou les colonnes du `DataFrame` de gauche sur lesquels faire le jointure, et le paramètre `right_on` propose le comportement analogue pour le `DataFrame` de droite.

Le paramètre `how`, dont les valeurs possibles sont `left`, `right`, `inner` et `outer`, permettent de faire des jointures à gauche, à droite, intérieures et pleines (par défaut, c'est la valeur `left` qui est utilisée).

```
In [ ]: def join(
self,
other: DataFrame,
left_on: List[str] | str,
right_on: List[str] | str,
how: str = "left"
) -> DataFrame
```