# INTRODUCTION

## à l'**O**bject **O**riented **P**rogramming (OOP) en Python

# Objectif global de l'activité

- L'objectif de ces trois jours est de vous familiariser avec les grands concepts de la programmation orientée objet en Python (et de manière plus générale, à la programmation orientée objet indépendamment du langage utilisé) : **classes, objets, instances, attributs, méthodes, héritage, polymorphisme, encapsulation, etc.**
- Cette initiation à l'**OOP** devrait vous permettre de mieux comprendre la manière dont sont structurées certaines librairies que vous manipulez régulièrement.

# Contraintes

Pour changer un peu, et pour vous familiariser avec d'autres manières de concevoir des programmes informatiques nous n'allons pas utiliser de Jupyter Notebook pour ce module. Vous écrirez votre code dans **VS Code**[1] (ou PyCharm) et vous exécuterez votre code en ligne de commande (CLI pour *Command Line Interface*).

# Overview et modalités du travail à réaliser

Le projet est découpé en deux projets :

1. **Projet Zoo**
   - Ce projet a pour but de passer en revue quelques-uns des grands concepts de l'**OOP**, et de les manipuler.
   - Cette partie du projet est à réaliser seul.e ou en binôme.
   - Idéalement le projet est à réaliser sur **½ journée à 1 journée max** pour vous laisser le temps d'aborder le projet Puissance 4.
   - Si vous vous retrouvez en difficulté face à l'OOP en Python vous pouvez y passer plus de temps (au détriment du projet Puissance 4)

2. **Projet Puissance 4**
   - Cette partie du projet est à réaliser en groupe : 2 à 3 personnes.
   - Ce projet est significativement plus difficile que le projet Zoo.
   - En cas de difficulté trop grande pour le démarrage du projet, une base de code vous sera fournie pour la visualisation du plateau.

**Compétences**

Les trois compétences du module ne se basent que sur la partie Zoo.

**Formateur** : Vincent Faure

[1] https://code.visualstudio.com/Download

# Projet Zoo

## Objectif

L'objectif de cette première partie est d'assimiler les grands concepts de l'OOP au travers d'un cas d'étude hyper simplifié : un **zoo.**

## Compétences associées à l'activité

- Créer une classe, l'instancier et faire appel à ses méthodes
- Utiliser les notions d'héritage et de polymorphisme
- Utiliser la notion d'association entre classes

## Démarche pédagogique

- Mise en œuvre progressive des principales notions de programmation orientée objet.
- Chaque notion est présentée par une brève description, puis détaillée par le biais d'une ou plusieurs ressources. À chaque notion est associée une tâche à effectuer.

## Consignes

- Etude de tutoriels
  - Introduction à l'OOP en Python (en français) :
    - https://www.pierre-giraud.com/python-apprendre-programmer-cours/
    - Parties 19 à 22 de ce cours.

- Afin de mettre rapidement en application ces connaissances et illustrer certains concepts nous vous proposons de créer un "zoo" virtuel en python.
- Étapes de la construction du zoo:

  - **[Classe]** Créer une classe *`Animal`* qui possède un **constructeur** (prenant en paramètre un poids et une taille)

  - **[Méthodes]** Ajoutez une méthode *`se_deplacer( )`* qui pour le moment ne fait rien.
    - **Ressource** :
      - https://docs.python.org/fr/3/tutorial/controlflow.html#pass-statements

  - **[Instanciation et attributs]** Dans le programme principal, **instanciez** cet objet. Faites un print de ses **attributs**.

  - **[Héritage]** Créez maintenant deux autres classes : une classe *Serpent* et une classe *Oiseau*. Ces deux classes **héritent** de la classe *`Animal`*.*

  - **[Polymorphisme]** Chacune de ces deux classes **redéfinira** la méthode *`se_deplacer( )`* : un serpent pourra par exemple faire un print *"je rampe"* alors que l'oiseau fera un print *"je vole"*. Instanciez maintenant ces deux objets dans le programme principal et appelez la méthode *`se_deplacer( )`* pour chacun de ces objets.
    - **Ressource** : https://www.edureka.co/blog/polymorphism-in-python/#polymorphism

  - **[Mot clé "super"]** Vous pouvez étendre et/ ou modifier le comportement de certaines méthodes des classes enfants qui héritent d'une autre classe grâce au mot clé "super". Nous choisissons maintenant de rajouter un nouvel attribut chez l'oiseau : l' **`altitude_max`** de vol. Servez-vous maintenant du mot-clé super pour redéfinir la fonction **`__init__`** chez l'oiseau sans avoir à tout réécrire.
    - **Ressource** : https://realpython.com/python-super/

- **[Encapsulation]** Afin de protéger certaines des données et de pouvoir éventuellement effectuer une vérification avant de les modifier, passez les attributs de la classe en "***protected***" (ou **"private"**) et accédez aux données via des **getters et setters**. Ajoutez une validation de données dans le setter, vous pouvez par exemple traiter le cas de l'instanciation d'un animal avec un poids négatif : cette tentative doit par exemple renvoyer une erreur (`ValueError`).
  - **Ressources** :
    - Getters and setters :
      - https://www.datacamp.com/community/tutorials/property-getters-setters
    - Exceptions
      - https://docs.python.org/3/library/exceptions.html#ValueError
      - https://docs.python.org/fr/3/tutorial/errors.html#handling-exceptions)

- **[Association]** Créez maintenant une nouvelle classe que vous appellerez **`Zoo`**.
  - Cette classe possède un attribut qui est une liste d'objets de type **`Animal`**. Vous passerez la liste d'animaux comme paramètres lors de l'instanciation de l'objet.
  - Vous définirez une méthode permettant d'ajouter un objet de type **Animal** à une instance de **Zoo**.

- **[Operator overloading]** Vous pouvez également redéfinir certains opérateurs au sein des classes pour qu'elle réponde à vos besoins.
  - Commencez par redéfinir la méthode **`__str__`** de la classe **`Animal`** pour qu'à l'appel de la fonction **`str`** soit retourné un string définissant de manière plus explicite l'objet (après avoir instancié un objet Animal que vous appellerez **"ani"** → exécutez **`print(str(ani))`** pour voir ce qui est affiché).
  - On peut aussi **surcharger** les opérateurs **">"** , **"<" et "==", "+"**, etc. À titre d'exemple nous allons maintenant rajouter un support pour l'opération "+" entre objets de type Zoo.
    - Débrouillez vous pour que l'ajout de deux objets de type **Zoo** (ie. z3 = z1 + z2, avec z1 et z2 objets de type Zoo) retourne un objet de type Zoo dont l'attribut est la liste concaténée des deux objets.
  - **Ressources** :
    - https://levelup.gitconnected.com/python-dunder-methods-ea98ceabad15
    - https://docs.python.org/3/reference/datamodel.html#special-method-names
    - https://www.programiz.com/python-programming/operator-overloading