

Note : Ce support sera complété et amené à évoluer au fur et à mesure des séances (dernière version disponible sur Moodle).

Web Full Stack

par Alexandre CASES

Dernière mise à jour le 20/10/2023





1. Introduction





Introduction

Description de la matière :

Approche des technologies pour devenir un développeur full Stack. Son développement utilisera un framework 'tout en un' capable de propulser aussi bien la partie backend que l'interface utilisateur dans le navigateur (ex : Ruby On Rails, Django, ASP.Net Core).

Compétences à valider :

- Maîtriser les fondamentaux du Framework choisi
- Créer rapidement des applications Web complexes
- Déployer une application sur un serveur
- Maîtriser le design pattern MVC
- Sécurisation d'une application web complexe (XSS, Anti forgery token, ...)



Tour de table

- Expérience de développement ?
 - Technologies maîtrisées / utilisés ?
 - Connaissance de Python ?
 - Besoin d'une petite mise à niveau avant d'aller sur du Django ?
 - Attentes sur le module ?
 - Etc...
-
- Retour de fin de séance (optionnel) : <https://forms.gle/2hgxTTXvffj4xY8d6>

Ce lien sera valable tout au long du module.



Déroulement et évaluations

Déroulement (10 séances) :

- Partie cours théorique
- Mise en oeuvre sur des cas pratiques
 - Technologie de référence utilisé pour le cours : Python (framework Django)
- Projet fil rouge
 - Technologie au choix tant qu'elle répond au besoin

Evaluations :

- TP et/ou Devoir et/ou QCM de fin de module
- Projet

⇒ Le but de étant d'avoir l'opportunité de valider les compétences.



Rappel degré de maîtrise

Non évaluable	Non maîtrisé	Maîtrise partielle	Bonne maîtrise	Excellente maîtrise
Compétence non mise en œuvre	Agit sans méthodologie ou avec une méthodologie inadaptée	Met en œuvre une méthodologie incomplète	Comprend et met en œuvre une méthodologie rigoureuse	Propose une méthodologie pertinente
	N'utilise pas d'outils ou ne les maîtrise pas	Mobilise correctement quelques outils	Choisit les outils adaptés	Adapte et/ou élabore des outils opérationnels
	N'analyse pas	Analyse de manière incomplète	Analyse de manière pertinente	Analyse et remédie
	Communique de façon non appropriée	Rend compte sans argumentation	Argumente et fait comprendre	Fait adhérer
	N'utilise pas d'information	Utilise partiellement les informations	Recherche et mobilise l'information	Produit des informations pertinentes et exploitables
	N'atteint pas les objectifs	Atteint en partie les objectifs	Atteint les objectifs	Dépasse les objectifs
	Ne formule aucune proposition ou des propositions incohérentes	Formule quelques propositions	Justifie et argumente ses propositions	Est force de proposition
	Ne prend pas en compte les contraintes	Prend en compte partiellement les contraintes	Intègre l'ensemble des contraintes	Anticipe les contraintes



2. Projet fil rouge





Séance 1 : Choix du projet et présentation

- Sujet au choix : Sujet fil rouge Ynov (à privilégier), Sujet perso ou Proposition de sujet

⇒ 6 groupes (27 étudiants) : 3 groupes de 5 et 3 groupes de 4

- Petite présentation du sujet (~10 min) :
 - Contexte
 - Titre du projet
 - Introduction / Présentation
 - Description courte du projet
 - Comment le sujet permettra de mettre en place les compétences à valider
 - Technologies choisi (avec justification)
 - Identification des parties prenantes
 - Liste des exigences liées (S'appuyer sur les PP)
 - Spécifications fonctionnelles (contenu au choix : (BPMN, US, UC, Kanban, etc...))
 - Maquettage
- Selon temps, avancer la suite :
 - Création des dépôts github
 - Outils de gestion de projet
 - Etc...

Séance 2 & 3 : Sprint 0

1. Sprint 0 : Découpage du projet / Gestion du Backlog

- Découpage des fonctionnalités (Trello ? Jira ? ...)
- Estimations des US (par complexité !)
- Budgétisation du projet (TJM, gestion, marges, frais annexes: hébergement par ex) justifiée (v1 – v2 - vN ?) Ressources externes ?
- Planification et engagement du Sprint 1
- Démo avancement
- Questions / Réponses

2. Selon le temps :

- Avancer la partie technique du projet

Prévoir présentation de 10min à la fin de séance 3 (+/- 2min)

Séance 4 & 5 : Sprint 1

1. Sprint 1

- Avancement du sprint (qu'est ce qu'on a fait...)
- Démo (jeu de données et scénario préparé)
- Planification et engagement du Sprint 2
- Questions / Réponses

2. Selon le temps :

- Avancer la partie technique du projet

Prévoir présentation de 5 min (+/- 1min)

Séance 6 & 7 : Sprint 2

1. Sprint 2

- Avancement du sprint (qu'est ce qu'on a fait...)
- Démo (jeu de données et scénario préparé)
- Planification et engagement du Sprint 3
- Questions / Réponses

2. Selon le temps :

- Avancer la partie technique du projet
- Avancer la partie Présentation finale (cf slide 13)
-

Prévoir présentation de 5 min (+/- 1min)

Séance 8 & 9 : Sprint 3

1. Sprint 3

- Faire la partie Présentation finale (cf slide 13)

2. Selon le temps :

- Avancer la partie technique du projet

Projet - Présentation finale

1. **Contexte général du projet** (*Contexte général, fonctionnalités principales...*)
2. **Gestion de projet** (*Description des sprint, organisation équipe, methodo...*)
3. **Spécifications fonctionnelles** (*Schémas, UML, US...*)
4. **Spécifications techniques** (*Schémas, UML, US...*)
5. **Présentation rapide du code source (essentiel et pertinent)**
6. **Démo (*jeu de données et scénario préparé*)** (*On montre les fonctionnalités, on vend du rêve*)
7. **Bilan du projet** (*humain, technique, avantages, inconvénients, avis perso,...*)
8. **Pour la suite du projet ?** (*Reste à faire, idées, etc...*)
9. **Questions / Réponses** (*Groupe et Individuel*)

Note : Ceci est un fil rouge, libre à vous de l'adapter et d'ajouter des éléments pertinents à votre présentation.

Chaque groupe devra faire une présentation de **10-15 min**

Chaque groupe jouera le rôle du client (*remarques, compréhension, demandes supplémentaires, ...*)



3. Cours théoriques





3.1. Python - Les bases





3.1.1. Introduction Python



Introduction

- Langage de programmation créé par **Guido van Rossum** en **1991**
- Facile à **apprendre** et **maintenir**
 - Peu de contraintes liées à la syntaxe
 - Typage faible / dynamique
- **Gestion automatique** de la mémoire et **gestion** des **exceptions**
- Python est un langage **interprété** □ **Interpréteur** adapté nécessaire (*pour exécuter le code*)
- Langage Orienté Objet □ **POO**

Versions



- Le langage Python **évolue** constamment, on parle alors de **version**

- Les versions **majeures** sont :
 - **1994**: Version **1.0**
 - **2000**: Version **2.0**
 - **2008**: Version **3.0**
 - **2022**: Version **3.10.4** (version actuelle)

- Attention ! Certaines **bibliothèques** ou **modules** requiert des versions spécifiques !

Syntaxe et indentation

- En python, la **syntaxe** est très importante (mais peu contraignante)
- On parle **d'exécution séquentielle**, **d'indentation** et de **bloc**
- Aucune préoccupation de **délimiteur** de bloc (*comme Java, JS, ...*)
- Un programme n'est exécuté que si **TOUTES** les instructions sont correctes

Prérequis



- Les prérequis pour faire de la programmation en Python sont :
 - Un **interpréteur** Python
 - Un environnement de développement adapté (**IDE**)

- Installation de l'interpréteur:
<https://www.python.org/downloads/> (ou *apt-get install python3*)

- Installation de l'IDE **Visual Studio Code** (et plugin Python) :
<https://code.visualstudio.com/download>

3.1.2. Les bases

Gestion mémoire (1/2)

- **Variable:** Nom + valeur
- **Mémoire:** Dispositif permettant de stocker l'information en machine
- Toutes les variables sont enregistrées en mémoire
- On parle de **cases mémoires**:
Adresse + Contenu

Adresse	1	2	3	4	5
Contenu	32	25		"hey"	
Adresse	6	7	8	9	10
Contenu			18		

Gestion mémoire (2/2)

- Python gère **automatiquement** l'insertion de la mémoire
- Pour lire et/ou récupérer une donnée, on récupère en réalité son **adresse mémoire**
- Exemple :

Adresse: 4
Variable: *myVarName*
Valeur: "hey"

Adresse	1	2	3	4	5
Contenu	32	25		"hey"	
Adresse	6	7	8	9	10
Contenu			18		

Règles de nommage (1/2)

- Le nommage d'une variable est autorisé avec les caractères suivants :

Lettres / Chiffres / Underscores / Tirets

- Les caractères interdits sont :

Accents / Signe de ponctuations / @ / Chiffre en premier

- Syntaxe et exemples : `<var_name> = <value>`

- A = 3
- a = "Bonjour"
- maVariable = True

Règles de nommage (2/2)

- En plus des règles précédentes, Python possède des **mots clés réservés** (qu'on ne peut donc pas utiliser) :

print
in
and
or
if
del
for

is
raise
assert
elif
from
lambda
return

break
else
global
not
try
class
except

while
continue
exec **eval**
import **pass**
yield

def
finally
async
Await
None
nonlocal

Type de données (1/2)

□ **Typage**: Indication sur la nature d'une variable □ `type(var_name)`

□ Python dispose de **4 types faibles** (ou primitifs) + **5 types supplémentaires** :

- Nombre: **int** (Integer)
- Nombre décimaux: **float** (Flottant)
- Chaîne de caractères: **str** (String)
- Vrai ou Faux: **bool** (Booléen)
- Liste: **list**
- Dictionnaire: **dict**
- Collection: **tuple**
- Nombre complexe: **complex**
- Absence de valeur : **noneType**

Type de données (2/2)

□ Exemples :

Integer	String	Float	Boolean
4	'Pierre'	4.50	True
-17	'a'	10.8	False
100000	'je suis un texte'	1.00	

Les commentaires

- ▣ **Les commentaires** permettent d'ajouter des informations complémentaires (*non interprété par le programme*)
- ▣ Il existe 2 types de commentaires (sur une ou plusieurs lignes)

```
1 print("Ceci est un code python très sérieux") #je peux commenter
2 #je peux commenter une ligne complète
3 print("On est de retour dans le programme")
4 """
5 Un commentaire
6 sur beaucoup
7 de lignes
8 """
9 print("et la encore")
```

Casting et concaténation

- Le **Casting** permet de modifier le **type** d'une variable
 - `str(5)` □ Le nombre 5 devient la chaîne de caractères "5"
 - `int('5')` □ Le String "5" devient le nombre 5
 - `float(5,6)` □ Le String "5,6" devient le float 5,6
 - `bool("True")` □ Le String "True" devient le booléen *True*
 - `int("test")` □ Impossible ! □ Génère une erreur !
- La **Concaténation** permet de cumuler plusieurs chaînes de caractères (*et/ou variables*) ensemble
- La concaténation s'effectue avec le symbole **+**
`a = "Je" b = "suis "` `a + b` □ "Je suis"

Casting et concaténation - Exemples

```
1  points = 3.6 # points est du type float
2  print("Tu as " + points + " points !") # Génère une erreur de typage
3
4  points = int(points) # points est maintenant
5  |         |         |         |         | #du type int (entier),
6  |         |         |         |         | #sa valeur est arrondie à l'unité inférieure(ici 3)
7  print("Tu as " + points + " points !") # Génère une erreur de typage
8
9  points = str(points) # points est maintenant
10 |         |         |         |         | #du type str (3 -> "3")
11
12 print("Tu as " + points + " points !") # Plus d'erreur de typage,
13 |         |         |         |         | #affiche 'Tu as 3 points !'
```

Les opérateurs arithmétiques

- Il est possible d'effectuer toutes **sortes d'opérations** sur des variables **numériques** (ou **boolean**) :

Opérateur	Nom
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo
**	Puissance
//	Division entière

Interactions H/M

- On différencie un **script** d'un programme **interactif**
- Un programme à pour but de créer des **interactions** entre ***l'Homme*** et la ***Machine***
- On a alors besoin de 2 types d'interactions (*qui s'appuie sur 2 fonctions natives Python*) :
 - Affichage dans la console : **méthode `print()`**
 - Récupération saisie clavier : **méthode `input()`**

La fonction Print()

- La Fonction **print()** permet d'afficher du texte dans la console
- Le texte peut être **simple** ou **formatée**
- Une **écriture formatée** permet d'insérer des variables dans une chaîne de caractère, on utilisera pour cela des **f-strings** (depuis la v 3,6) :
- Syntaxe:
`print(f"mon text {maVar}")`

La fonction Input()

- La Fonction **Input()** permet de **récupérer des informations** saisies quand le programme est lancé
- La fonction **Input()** ne récupère que des chaînes de caractères (**Strings**)
- Syntaxe:
`maVar = input("Texte à afficher")`

Exemples

PRINT

```
a = 'Bonjour'
b = 5
c = 6

print(a)
print(b+c)
print(c*3)
print(b, c, sep=' // ')
print(f"Les valeurs de b et c sont {b} et {c}")
```



```
Bonjour
11
18
5 // 6
Les valeurs de b et c sont 5 et 6
```

INPUT

```
myValue = input("Saisir un mot: ")
print(myValue)
print(type(myValue))
```



```
Saisir un mot: ALO
ALO
<class 'str'>
```

Création et exécution du programme

- Un programme Python est identifiable avec son extension: **.py**
- Pour **exécuter** un programme, les syntaxes (*selon la version installée*) sont les suivantes :

py programme_name.py

python programme_name.py

python3 programme_name.py

Exercice 1

1. Créer un premier programme (*tp1.py*)
2. Instancier 3 variables (*a*, *b*, et *c*) avec les valeurs respectives 3, 12 et 4
3. Afficher un message (*Les trois variables sont ...*) + *leurs valeurs*
4. Créer une variable qui calcule la somme de *a*, *b* et *c*
5. Afficher un message (*La somme est...*) + *résultat*
6. Trouver un moyen d'inverser les valeurs de *a* et *b* (*a = b* et *b = a*)
7. Afficher un message (*a vaut maintenant (12) et b vaut (3)*)

3.1.3. Conditions et Boucles

Ordres d'exécution

- Un programme s'exécute selon un **ordre d'exécution**
- Il existe 2 type d'exécution en Python: **Séquentielle** et **Conditionnelle**

Séquentielle

```
a = 2  
  
b = 5  
  
c = a + b
```

Conditionnelle

```
a = 3  
  
if (a > 10):  
    print("Test")
```

Les conditions IF - ELIF - ELSE

- La structure de contrôle **IF/ELSE** permet d'ajouter une **condition** et de proposer plusieurs traitements différents.

```
a = 10

if (a < 10):
    print("Inférieur à 10 !")

elif (a == 10):
    print("Egal à 10 !")

else:
    print("Supérieur à 10 !")
```


Les boucles

- Les conditions de boucles permettent de **répéter N fois** un traitement.
- Il existe 2 façon de créer des boucles (**while** ou **for...in**)

```
a = 0
while (a < 5):
    print(a)
    i += 1
```

```
for a in range (1, 5):
    print(a)
```

```
for a in [1, 2, 3, 4, 5]:
    print(a)
```

Le mot-clé break

- Il est possible de **cumuler** et **imbriquer** plusieurs conditions/boucles
- Le mot clé **break** permet de sortir prématurément d'une boucle

```
for a in [1, 2, 3, 4, 5]:  
    if (a == 3):  
        break
```

Les opérateurs logiques

- Pour effectuer des **comparaisons** ou **conditions**, il faut utiliser les **opérateurs logiques** :

<code>==</code>	□ Egal à
<code>!=</code>	□ Différent de
<code><</code>	□ plus petit que
<code>></code>	□ plus grand que
<code><=</code>	□ plus petit ou égal à
<code>>=</code>	□ plus grand ou égal à
<code>or</code>	□ OU
<code>and</code>	□ ET
<code>not</code>	□ Non

```
a = 10
```

```
if (a <= 10):
```

```
    print("Inférieur ou égal à 10 !")
```

```
elif (a <= 20 and a >= 10):
```

```
    print("Entre 10 et 20 !")
```

```
elif (not a == 20):
```

```
    print("Supérieur à 20 !")
```

Exercice 2

1. Reprendre l'exercice précédent en demandant à l'utilisateur d'entrer les 3 valeurs a , b et c
2. Trouver la plus petite et la plus grande valeur entre a , b et c .
3. Afficher le min et le max

Exercice 3

Créer le jeu du nombre mystère. Le but du jeu est de trouver un nombre aléatoire (*entre 1 et 100*) en un minimum de tentative. A chaque essai (*raté*), le programme donne une indication pour savoir si le nombre à trouver est plus grand ou plus petit.

A la fin, le programme doit dire combien d'essais ont été réalisés pour trouver le nombre mystère.

Aide: Librairie **Random** à importer avec la méthode `randint(start, stop)`

```
import random  
  
myster = random.randint(0, 100)
```



3.1.4. Les Itérables



Qu'est ce qu'un Itérable ?

▣ **Itérable**: Objet regroupant un **ensemble** de valeurs.

▣ En Python, il existe 5 types d'itérable:

- **str** Les chaîne de caractères
- **list** Les listes
- **dict** Les dictionnaires
- **tuple** Les tuples
- **set** Les ensembles

La chaîne de caractères

- La **chaîne de caractère** en Python est un type à part
- Le type **str** est composé de **n** caractères
- Exemple :

```
my_char = "Ma chaîne!"  
for e in my_char:  
    print(e)
```



```
M  
a  
  
c  
h  
a  
i  
n  
e  
!
```


Les listes

- ▣ **Liste:** Collection ordonnée contenant une série de **valeur/donnée** dans un ordre précis (**index**)
- ▣ Une liste en Python peut contenir des **types différents**
- ▣ Il est possible d'accéder à n'importe quel élément dans une liste à l'aide de son **index** (Premier index => **0**)
! Attention ! L'accès à un index n'existant pas provoque une erreur !
- ▣ Syntaxe:
`ma_liste = [val1, val2, valN]`

Quelques exemples

```
villes = ['Toulouse', 'Paris', 'Lyon']  
print(villes[1])
```

```
# Result => Paris
```

```
nombres = [15, 33, 20, 15978, 6]  
print(nombres[0])
```

```
# Result => 15
```

```
melange = [153, 'Bonjour', 0.8, True]  
for e in melange:  
    print(str(e) + " est de type " + str(type(e)))
```

```
# Result => 153 est de type <class 'int'>, Bonjour est de type <class 'str'>  
#           0.8 est de type <class 'float'>, True est de type <class 'bool'>
```

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

```
# Result => [4, 5, 6]
```

Quelques méthodes

Méthodes	Exemples
Initialisation	<code>my_list= []</code> / <code>my_list = [1,2,10, 5]</code>
Index / Accès	<code>my_list[0] □ 1</code> / <code>my_list[-1] □ 5</code>
Longueur	<code>len(my_list) □ 4</code>
Tri, min, max	<code>my_list.sort() □ [1, 2, 5, 10]</code> / <code>max(my_list) □ 10</code>
Insertion	<code>my_list.append(99) □ [1, 2, 5, 10, 99]</code>
Insertion (position)	<code>my_list.insert(1, 66) □ [1, 66, 2, 5, 10, 99]</code>
Suppression	<code>my_list.pop() □ [1, 66, 2, 5, 10]</code>
Recherche (in)	<code>99 in my_list □ True</code> / <code>98 in my_list □ False</code>

Exercice 4

1. Créer un programme qui demande à l'utilisateur d'entrer des nombres jusqu'à ce que l'utilisateur tape la lettre 'q' (comme quitter).
2. Ajouter chaque nombre saisi dans une liste, en excluant les doublons.
3. Quand l'utilisateur quitte, le programme génère une liste de tous les nombres que l'utilisateur a saisi en ayant supprimé les doublons.

Exercice 5

1. Écrire un programme qui analyse un par un tous les éléments d'une liste de prénom.
2. Créer deux listes "*small_name*" et "*long_name*", qui comprendrons respectivement les mots comportant moins de 6 caractères et 6 caractères ou plus.
3. Afficher à la fin du programme les 2 listes et leur nombres d'éléments respectifs

Exemple de liste: [*Jean*, *Maximilien*, *Brigitte*, *Sonia*, *Abdelkarim*, *Sandra*]

Les Tuples

- ▣ **Tuple:** Collection ordonnée de plusieurs éléments
- ▣ Un tuple ne peut être modifié après sa création
- ▣ Un tuple peut-être utile pour **retourner** plusieurs valeurs d'une fonction
- ▣ Syntaxe:

`mon_tuple = (val1, val2, valN)`

```
mon_tuple = (1,4,6,5)
mon_tuple[0]      # 1
max(mon_tuple)    # 6
4 in mon_tuple    # True
```

Les Dictionnaires

- Un dictionnaire est un **ensemble d'élément non trié**
- Il n'a pas **d'index** pour identifier une valeur, mais une **clé**
- On parle d'ailleurs de couple **clé/valeur**
- Syntaxe:
`mon_dict = {"key1": "value1", "key2": "value2", "keyN": "valueN"})`

Quelques méthodes

Méthodes	Exemples
Initialisation	<code>my_dict = { }</code> / <code>my_dict = {"nom": "Dubois", "age": 22 }</code>
Accès	<code>my_dict["nom"]</code> □ « Dubois » / <code>my_dict["a"]</code> □ ERROR <code>my_dict.get("nom")</code> □ « Dubois » / <code>my_dict.get["a"]</code> □ None
Longueur	<code>len(my_dict)</code> □ 2 <i>renvoie la longueur de Tuple !</i>
Insertion / Modification	<code>my_dict["prenom"] = "Joe"</code> <code>my_dict["age"] = 23</code>
Suppression	<code>del my_dict["prenom"]</code> □ Supprime le couple clé/valeur <code>my_dict.clear()</code> □ Vide le dictionnaire
Parcourir (<i>en liste</i>)	<code>my_dict.keys()</code> □ ['nom', 'age'] <code>my_dict.values()</code> □ ['Dubois', 22] <code>my_dict.items()</code> □ [('nom', 'Dubois'), ('age', 22)] □ TUPLE

Les Dictionnaires – Exemple

```
my_dict = {}  
my_dict["value1"] = "Ma premiere valeur"  
my_dict["value2"] = "Ma deuxième valeur"  
my_dict["valueX"] = "Ma valeur X"  
  
for k, v in my_dict.items():    # Tuple  
    print("Clé", k)  
    print("Value", v)
```



```
Clé value1  
Value Ma premiere valeur  
Clé value2  
Value Ma deuxième valeur  
Clé valueX  
Value Ma valeur X
```

Exercice 6

Ecrire un programme qui transforme une liste de température (*type int*) en dictionnaire de correspondance

Température/Nombre de jour

A la fin du programme, parcourir le dictionnaire et afficher les résultats, trié par ordre croissant de température.

Exemple de liste:

Résultat attendu :

```
températureJuin = [  
    20, 13, 17, 15, 18, 19, 20,  
    17, 17, 17, 18, 10, 22, 18,  
    17, 21, 15, 16, 16, 14, 19,  
    18, 15, 11, 12, 12, 16, 15,  
    16, 19  
]
```

```
10°C: 1  
11°C: 1  
12°C: 2  
13°C: 1  
14°C: 1  
15°C: 4  
16°C: 4  
17°C: 5  
18°C: 4  
19°C: 3  
20°C: 2  
21°C: 1  
22°C: 1
```



3.1.5. Fonctions et Procédures



Introduction

- **Procédure:** Bloc de code permettant d'exécuter une série d'instruction
- **Fonction:** Bloc de code permettant d'effectuer des traitements et de **retourner** une ou plusieurs valeurs
- Dans les deux cas, elles sont utilisées pour **éviter les répétitions**
- Une fonction et une procédure acceptent entre 0 et N **paramètres**

Syntaxe

- La syntaxe pour écrire des **procédures** et **fonctions** est proche: Il faut utiliser le mot-clé **def**
- La différence entre les deux est qu'une fonction **renvoie une valeur** (mot clé **return**) et une procédure ne renvoie rien

- Syntaxe :

```
def <function_name> (arg1, arg2, argN):  
    # instructions ou traitements  
    return <value>      (SI FONCTION)
```

Exemples

```
# Procédures
def procedure_test_1():
    print("Lancement de la procedure 1")

def procedure_test_2(x):
    print("Lancement de la procedure 1 avec le param", x)

procedure_test_1()
procedure_test_2('blabla')
```

```
# Fonctions
def function_test_1():
    return 'Lancement de la procedure 1'

def function_test_2(x, y):
    print("Lancement de la procedure 2 avec les params", x, y)
    return "OK"

def function_test_3():
    return 3, 9, True

print(function_test_1())
z = function_test_2('blabla', 4)
print(z)
a, b, c = function_test_3()
print(a, b, c)
```

Variables locales

- Les variables utilisées à l'intérieur d'une fonction/procédure **n'existent que** dans la fonction/procédure
- **Attention !** Pour les **listes** et les **dictionnaires**, les modifications s'effectuent également en dehors de la procédure

Variables locales - exemples

```
a = 3
b = [3, 45, 11]
c = {'a': 4, 'b': 10}

def test1(v):
    v = 100

def test2(l):
    l.append(999)

def test3(d):
    d['c'] = 77

test1(a)
test2(b)
test3(c)
print(a, b, c)
```



```
3
[3, 45, 11, 999]
{'a': 4, 'b': 10, 'c': 77}
```


Exercice 7

1. Demander à l'utilisateur d'entrer un nombre entre 1 et 3
2. Créer la **procédure** *choix1()* qui affiche dans la console « *Le choix est 1* »
3. Créer la **fonction** *autreChoix(p)* avec *p* correspondant à la saisie de l'utilisateur
 - La fonction doit retourner **True** si le nombre est égal à 2 ou 3.
 - La fonction doit retourner **False** sinon
4. Ajouter une condition selon le retour de la fonction :
 - Si c'est True, afficher « *Le choix est +choixSaisi* »
 - Si c'est False, afficher « *Votre choix est en dehors de l'intervalle !* »

Exercice 8 (suite 6)

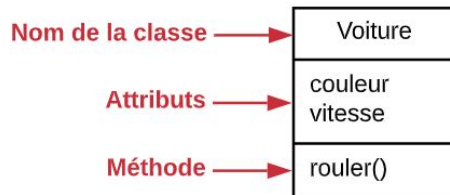
1. Créer un programme qui demande à l'utilisateur d'entrer les températures du mois, à renseigner dans une liste.
2. Dès qu'une des deux conditions (lettre 'q' saisie **ou** 31 températures saisies) est remplie, proposer à l'utilisateur un menu, avec les possibilités suivantes:
 - Afficher la température maximale
 - Afficher la température minimale
 - Afficher la température moyenne ☐ fonction **moyenne()**
 - Afficher toutes les températures par ordre croissant ☐ procédure **liste()**
 - Quitter le programme
3. Afficher dans la console le résultat de son choix et proposer de nouveau les différentes possibilités (tant que l'utilisateur n'a pas quitter le programme)

Il faut écrire la **procédure** `liste()` et la **fonction** `moyenne()`

3.1.6. La POO

Introduction

- La **programmation objet** est très utile dans le développement logiciel
- La POO permet de **créer** ses propres types et **structurer** son programme
- On parle **d'objet** (ou **classe**) et **d'instance de classe**
- Un objet est défini par un **nom**, des **attributs** et des **méthodes**



Notions importantes (1/2)

- La déclaration d'un objet s'effectue à l'aide du mot clé **class** et du **nom** de l'objet correspondant
- Les **attributs** correspondent aux **propriétés** de l'objet et s'ajoutent (par convention) juste après le mot clé class
- Les **méthodes** correspondent aux **fonctionnalités** propres de l'objet et s'ajoutent (par convention) juste après les attributs
- Un **constructeur** est une méthode d'initialisation d'un objet. Le constructeur s'écrit avec la syntaxe suivante :

```
def __init__(self):
```

Notions importantes (2/2)

- Le mot clé **self** fait référence à l'objet courant
- Il est possible de créer autant **d'instance d'objet** que souhaité en appelant le nom de l'objet correspondant
- Pour instancier un objet, il faut appeler la **référence** de l'objet souhaité

Pour résumer !

- Pour définir une classe, on utilise le mot clé **class**
- Pour créer une instance de classe, on utilise **className()**
- Pour définir des méthodes, on utilise **def methodeName():**
- Pour appelé l'objet courant, on utilise le mot clé **self**
- Pour ajouter un constructeur, on utilise **def __init__(self):**

Remarque : Les attributs de classe ne sont pas fixe en Python (Par convention et BP, il est préférable de les ajouter)

Examples

```
class MaClass1:
    id = 0
    text = None

    def __init__(self, id=0, text=None):
        self.id = id
        self.text = text

    def maMethode(self):
        print(self.text)

a1 = MaClass1()
a1.id = 1
a1.text = "A1"
a2 = MaClass1(2, "A2")

print(a1, a2)
print(a2.id)
a2.maMethode()
```

```
class MaClass2:
    id: int
    text: str

    def __init__(self, pId, pText):
        self.id = pId
        self.text = pText

maclass = MaClass2(5, "TEXT-TEST")

print(maclass.id)
print(maclass.text)
```

5
TEXT-TEST

```
<__main__.MaClass1 object at 0x00000263FB31FFD0> <__main__.MaClass1 object at 0x00000263FB31FF10>
2
A2
```


Les énumérations (1/2)

- En Python, il existe un type de donnée qui permet de lister des **constantes**: Les **Énumérations**
- Une **énumération** se crée comme n'importe quelle classe, en y ajoutant l'héritage de la classe **Enum**
- Dans une énumération, chaque constante à un **code** (toujours en majuscule par convention !) et une **valeur**
- Le seul prérequis pour utiliser des énumérations est **d'importer** le module **Enum**:
`from enum import Enum`

Les énumérations (2/2)

- Une énumération permet de manipuler des chaînes de caractères **prédéfinies** grâce à leur **code** d'identification
- Il est possible **d'accéder** aux différents éléments d'une énumération:
 - Son **Code** □ MonEnum.monCode.**name**
 - Sa **Valeur** □ MonEnum.monCode.**value**
- A l'inverse, il est possible d'accéder aux énumérations:
 - Grâce au **code** □ MonEnum['myValue']
 - Grâce à la **valeur** □ MonEnum(name)

Les énumérations – Exemple 1

```
from enum import Enum

class Couleur(Enum):
    NOIR = "Noir"
    BLANC = "Blanc"
    VIOLET = "Violet"

print(Couleur)
print(Couleur.NOIR.name)
print(Couleur.BLANC.value)

print(Couleur['VIOLET'])
print(Couleur('Noir'))
```



```
<enum 'Couleur'>
NOIR
Blanc
Couleur.VIOLET
Couleur.NOIR
```

Les énumérations – Exemple 2

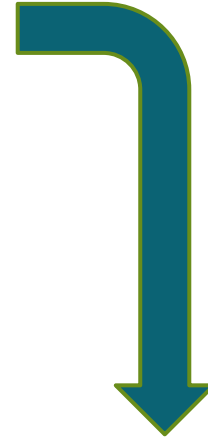
```
from enum import Enum

class State(Enum):
    STRATED = "Démariné"
    FINISHED = "Terminé"
    UNKNOW = "Non identifié"

list_etats = []

for s in ["Démariné", "Démariné", "Terminé", None]:
    if s == State.STRATED.value or s == State.FINISHED.value:
        list_etats.append(State(s))
    else:
        list_etats.append(State.UNKNOW)

print(list_etats)
```



```
[<State.STRATED: 'Démariné'>, <State.STRATED: 'Démariné'>, <State.FINISHED: 'Terminé'>, <State.UNKNOW: 'Non identifié'>]
```

Exercice 9

1. Créer un objet **Entreprise** avec 3 attributs (*nom*, *nbEmployes* et *CA*) et un constructeur.
2. Demander à l'utilisateur de renseigner un nom, un nombre d'employés et un chiffre d'affaire de son entreprise à créer.
3. Créer une instance d'objet avec les informations renseignées par l'utilisateur.
4. Afficher les informations de l'entreprise créée en s'appuyant sur l'instance d'objet créée, comme l'exemple ci-dessous :

L'entreprise MySociety a 551 employées à son actif et un chiffre d'affaires de 153098 euros

Exercice 10

Reprendre l'exercice précédent.

1. Au début du programme, demander à l'utilisateur le nombre d'entreprise qu'il veut créer.
2. Demander (pour le nombre d'entreprise à créer) à chaque fois le nom, nombre d'employé et chiffre d'affaire + Créer une instance d'objet + Ajouter les entreprises dans une liste.
3. Créer une méthode **display()** dans l'objet Entreprise qui retourne les informations d'une entreprise
4. A la fin du programme, parcourir la liste et appeler la méthode display() pour chaque instance d'Entreprise.

Structurer son code

- Pour avoir un code structuré, on préférera **diviser** un projet en plusieurs classes:
- Un programme ressemblera alors à la structure suivante :
Main + N Sous programme + Modèles
- Pour importer une classe « *models* », on utilise la syntaxe **import models** dans la classe appelante
- Il est possible de définir un **alias** avec le mot clé **as**

Exemple

```
# Fichier models.py
class MaClass1:
    id: int
    text: str

    def __init__(self, id=0, text=None):
        self.id = id
        self.text = text

    def maMethode(self):
        print(self.text)
```

```
# Programme principal
import models as m

a1 = m.MaClass1()
a1.id = 1
a1.text = "A1"
a2 = m.MaClass1(2, "A2")

print(a1, a2)
print(a2.id)
a2.maMethode()
```

```
<__main__.MaClass1 object at 0x00000263FB31FFD0> <__main__.MaClass1 object at 0x00000263FB31FF10>
2
A2
```


L'héritage (1/2)

- En Python, il est possible d'utiliser **l'héritage** de classe
- L'héritage permet **d'étendre** une (ou plusieurs) classe(s) existante(s)
- La sous classe **hérite** de tous les **attributs** et **méthodes** de la classe parente
- Pour hériter d'une classe existante, il faut définir dans les **paramètres** la **référence** à l'objet parent.

L'héritage (2/2)

- La classe fille peut donc hériter de la **méthode d'initialisation** de la classe mère en appelant sa méthode **init()**
- Le mot clé **super()** permet d'interagir avec la classe parente, et fait référence aux **attributs** de la classe parente.
- L'initialisation de la classe fille utilisera alors sa propre méthode d'initialisation en y intégrant celle de la classe mère à l'aide du mot clé **super()**

Exemple

```
class MaclassMere:
    id: int
    text: str

    def __init__(self, id, text):
        self.id = id
        self.text = text

class MaclassFille(MaclassMere):
    commentaires: str

    def __init__(self, id, text, coms):
        super().__init__(id, text)
        self.commentaires = coms

classM = MaclassMere(1, "maClasseMere")
classF = MaclassFille(2, "maClasseFille", "C'est mon premier commentaire !!")

print(classM.text)
print(classF.text)
print(classF.commentaires)
```



```
maClasseMere
maClasseFille
C'est mon premier commentaire !!
```

Exercice 11

Créer une classe **models** qui regroupe **une classe d'Enum** (*Micro, Petite, Grande*), **une classe Entreprise** (avec un attribut en plus pour le type d'entreprise qui s'appuie sur l'Enum) et **3 sous classes** (*qui héritent de Entreprise*).

Les 3 sous classes ont les valeurs suivantes par défaut à la création de chaque instance:

- ❑ **MicroEntreprise:** **type** = "Micro Entreprise"
- ❑ **PetiteEntreprise:** **type** = "PME" / **nbEmplie** = 6
- ❑ **GrandeEntreprise:** **type** = "Grande Entreprise" / **nbEmplie** = 50 / **montantCE:**
float

Reprendre le programme précédent et demander à chaque fois le type d'entreprise à créer et appeler la classe correspondante pour la création

Créer un dictionnaire avec 3 clés (=type Entreprise) et pour valeur une liste d'entreprise correspondante.

Parcourir le dictionnaire et afficher les entreprises à l'aide de la méthode display()



3.1.7. Gestion des exceptions



Gestion des erreurs

- Tout programme peut entrainer des **erreurs** pendant son exécution
- Un bon programme est un programme qui **anticipe** les erreurs **possibles** et les **traitent**
- Pour ce faire, Python met à disposition les **Exceptions**

Les exceptions

- Une **exception** est un **Objet** Python qui contient des **informations** sur le **dysfonctionnement** identifié
- Une exception non traitée entraîne **l'interruption** d'un programme
- Il existe plein d'exceptions différentes, dont les principales :
 - **ValueError** □ Valeur incorrecte (cas de casting par exemple)
 - **TypeError** □ Type incorrecte (opération, concaténation...)
 - **IndexError** □ Accès impossible (index de liste non existant)
 - **KeyError** □ Clé introuvable (clé de dictionnaire inexistante)
 - ...

Traiter les exceptions

- Pour traiter les exceptions, on utilise les mots clés **try** et **except**
- Le bloc `try` permet de **définir** la ou les instructions à tester
- Le bloc `except` permet de **recupérer** l'exception identifiée, ce bloc doit être **accompagné** du type d'erreur à traiter :
`except <errorType>:`
- Tout traitement exécuté dans un `try` est automatiquement **interrompu** dès qu'une erreur est identifiée
- Il est possible de **cumuler** plusieurs blocs d'exceptions dans un `try`

Information sur l'exception

- Toutes exception récupérée fournit un **objet** avec une **description** de l'erreur identifiée.
- La syntaxe pour afficher l'erreur est la suivante :

```
except <TypeError> as e:  
    print(e)
```

Else, Finally et Raise

- Il est possible d'ajouter un **bloc** pour **différencier** le bloc de code fonctionnel et non fonctionnel. Ce bloc sera exécuté **uniquement** si le bloc **try fonctionne**

Le bloc **else**

- Il est possible d'ajouter un **bloc** qui sera lu dans **tous les cas** après les blocs **try/except**

Le bloc **finally**

- Il est également possible **déclencher, signaler** et **customiser** une exception non pris en compte dans un try/except initial :

Le mot-clé **raise**

Exemples

```
def myFonction(x):  
    if x == 0:  
        raise ZeroDivisionError("Erreur - Division par zéro")  
    try:  
        chiffre = int(input("Entrez un chiffre : "))  
        chiffre /= x  
    except ValueError as e:  
        print("L'exception est levée !")  
    else:  
        print("Le traitement à fonctionnée !")  
    finally:  
        print("Le traitement est terminé !")  
  
# main  
try:  
    myFonction(1)  
    myFonction(0)  
except ZeroDivisionError as e:  
    print(e)
```

```
Entrez un chiffre : 10  
Le traitement à fonctionnée !  
Le traitement est terminé !
```

```
Entrez un chiffre : a  
L'exception est levée !  
Le traitement est terminé !
```

```
Erreur - Division par zéro
```

Exercice 12

Ecrire un programme qui à pour but de stocker les différentes notes d'un élève.

Au lancement du programme, le menu suivant apparait :

- Ajouter une note
- Afficher une note (en renseignant son index)
- Calculer la moyenne (à faire manuellement !)
- Quitter le programme

Faites en sorte de faire un programme qui ne peut pas planter. Vous devez implémenter au minimum les erreurs suivantes :

- ZeroDivisionError
- ValueError
- IndexError



3.1.8. Librairies et modules



Module

□ Un **module** est un fichier Python (extension py) qui contient un ensemble de **classes**, de **fonctions**, de **procédures** et de **variables** prédéfinies

□ L'intégration d'un module se réalise à l'aide des **imports**

□ La syntaxe pour ajouter un module dans la classe en cours est la suivante :

```
import <module_name>
```

□ **Toutes les fonctions** présentent dans le module importé sont utilisables avec la syntaxe si dessous :

```
<module_name>.<fonction_name()>
```

Alias et import limité

- Il est possible d'importer **tout le contenu** du module avec le mot clé *****
- Pour simplifier l'affichage, il est possible d'ajouter un **alias** avec le mot clé **as**
- Il est possible de ne cibler **que** la fonction à importer pour éviter d'importer **toute la librairie** avec le mot clé **from**
- Exemples :

```
>>> import toto
>>> toto.tata()
'tutu'
```

```
>>> from toto import tata
>>> tata()
'tutu'
```

```
>>> from toto import *
>>> tata()
'tutu'
```

Librairie (ou package)

- Un **package** (*ou librairie*) est une **collection** qui regroupe un ensemble de **modules** Python.
- Le module est un **fichier** et le package est **répertoire** de modules
- Un package contiendra toujours le fichier `__init__.py` □ Il s'agit d'un fichier essentiel pour Python

Modules courants

- On retrouve des **modules natifs** dans Python, dont les plus courants :
 - **random** : fonctions permettant de travailler avec des valeurs aléatoires
 - **math** : toutes les fonctions utiles pour les opérations mathématiques
 - **sys** : fonctions systèmes
 - **os** : fonctions permettant d'interagir avec le système d'exploitation
 - **time** : fonctions permettant de travailler avec le temps
 - **datetime** : fonctions permettant de travailler avec les dates
 - **profile** : fonctions permettant d'analyser l'exécution des fonctions
 - ...

Exercice 13

Créer un programme qui détermine aléatoirement deux données: un jour et un mois

Trouvez un moyen de transformer ces données en date (au format **YYYY-MM-DD HH:MM:SS**) et l'afficher

Renvoyer un print à la fin du programme pour connaître la différence de jour entre la date aléatoire et la date du jour. Définir également si la date aléatoire est passée ou non.

Pensez à gérer les exceptions

Aide: random et datetime

TP Personnage (part 1)

Personnage
nombreDeVies: int
nom: String
force: int point DeVie: int arme: String
parler()
frapper()

1. Créer la classe Personnage ci-joint, en implémentant ses attributs et un constructeur.
2. Implémenter la méthode '*parler()*' qui permet d'afficher un message à l'écran du type: « **perso1** dit : "**Je m'appelle Perso1**" » ou perso1 correspond à l'attribut nom et « Je m'appelle Perso1 » correspond à un message passé en paramètre de la méthode.
3. Implémenter la méthode '*frapper()*' qui à pour but de retirer des points de vie au personnage attaqué. Le personnage attaqué doit être identifié en paramètre de la méthode. De plus, la règle est la suivante :
 - Si ses points de vie sont inférieure ou égal à zéro, alors 1 vie doit lui être retiré et ses points de vie sont égale à 25.
 - Si son nombre de vie est égal à zéro, alors vous devez afficher un message : "*Le personnage <name_perso> est mort*"
4. Tester votre programme et vos méthodes !

Attention : Le programme doit avoir un main et des modèles séparés

TP Personnage (part 2)

1. Ajouter une classe d'énumération « Niveau » avec les trois possibilités, comme ci-joint.
2. Ajouter un attribut dans la classe Personnage pour renseigner le niveau du personnage.
3. Ajouter un menu de choix utilisateur dans votre programme, avec les possibilités suivantes :
 - i. Créer un Personnage: L'utilisateur peut renseigner le nom, la force et l'Arme du personnage à créer. Le nombre de vie est toujours égal à 2, les point de vie à 25 et le niveau à DEBUTANT. La création doit ajouter le nouveau Personnage dans une liste.
 - ii. Effectuer une Action: L'utilisateur peut effectuer l'action de frapper (en sélectionnant le Personnage à attaquer dans la liste) ou parler (en écrivant le message).
 - iii. Quitter le programme
4. La règle pour le niveau du personnage est la suivante :
 - Le niveau INTERMEDIAIRE augmente de 10% la force.
 - Le niveau EXPERT augmente de 25% la force et diminue de 10% les dégâts en cas d'attaque
 - Pour passer au niveau INTERMEDIAIRE, il faut que le personnage est réalisé 3 combats (attaque ou défense). Pour le niveau EXPERT, il faut 10 combat en tout.

<< Niveau >>
DEBUTANT
INTERMEDIAIRE
EXPERT

TP Personnage (part 3)

Créer le jeu correspondant avec les specs suivantes:

- Au lancement du jeu :
 - ❖ Combien de joueurs ?
 - ❖ Création des joueurs à tour de rôle (avec leurs propriétés)
 - ❖ Choix aléatoire de l'ordre des joueurs
- Pendant le jeu :
 - ❖ 1 action à choisir par joueur (menu avec possibilités)
 - ❖ 1 récap de l'état des joueurs
- Proposer une solution complète en V1
- Proposer une solution améliorée (Objet Arme, etc.) en V2



3.2. Python - Framework Django



Note importante

Tous les exercices seront fait sur un seul projet qui sera évolutif.



3.2.1. Introduction



Définitions

- **Framework**: Cadre de développement qui regroupe un **ensemble de fonctionnalités** prédéveloppées et documentées. Comparable à une bibliothèque XXL
- **API** (*Application Programming Interface*): **Interface** de communication entre plusieurs application
- **Django**: Framework Python permettant de mettre en place **rapidement** une application, une **API** associée et une **Base de Données**

Pourquoi Django ?

- Avantage du Framework :
 - Facile à implémenter et à maintenir
 - Rapide à prendre en main
 - Sécurisé
 - Multi DB
 - Beaucoup de documentation (et en français !)
- Le Framework **sépare les couches** en s'inspirant du modèle **MVC** (ou MVT)
- Django est également très **populaire** (aide en ligne, documentation...)

Historique

- ▣ **2003**: Début des développements du Framework
- ▣ **2005**: Première version au public
- ▣ **2008**: Fondation Django Software s'occupe de maintenir / faire évoluer le Framework
 - : Sortie de la première version stable
- ▣ **2017**: 2^{ème} version stable
- ▣ **2021**: Sortie de la version 4 (actuellement 4.2.6)



3.2.2. Dépendances et environnement virtuel



Prérequis

- L'utilisation de **Django** nécessite d'avoir un environnement adapté
- Il faut au minimum avoir l'interpréteur **Python** et la librairie **Django**
- Guide d'installation :
<https://docs.djangoproject.com/fr/4.1/intro/install/>
- Tests ?

```
py -V  
py -m django --version
```

Environnement virtuel - Intro

- Pour ajouter une **librairie externe** à Python, il existe 2 solutions:
 - Installer la librairie sur la machine physique
 - Utiliser un **environnement virtuel** pour un projet
- Le module **venv** donne la possibilité de créer un environnement virtuel (*léger, unique et segmenté*)
- Les Bonnes Pratiques préconisent de créer un environnement virtuel par projet
- Un environnement virtuel proposera également l'ajout de l'outil d'installation **pip** (*facilitateur d'installation*)
- Un fichier récapitulatif permettra de **lister** les librairies + versions: **requirements.txt**

Environnement virtuel - Manipulation

□ Utilisation de l'environnement virtuel et commandes associées :

1. Création de l'environnement virtuel: `py -m venv path/venv`
2. Utilisation/Activation du venv: `source path/venv/Scripts/activate`
3. Installation de librairie: `pip install <lib_name>`
4. Installation Django: `pip install Django`
5. Copie des librairies dans un fichier: `pip freeze > requirements.txt`
6. Installation des librairies depuis un fichier: `pip install -r requirements.txt`
7. Fermeture/Désactivation du venv: `deactivate`

<https://docs.python.org/fr/3/tutorial/venv.html>



3.1.3. Initialisation d'un projet

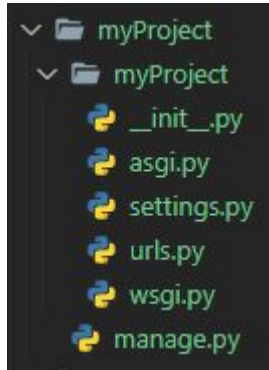


Commandes de création

- La Création d'une nouvelle application Django se fait en 3 étapes :
 1. Génération d'un nouveau Projet Django
 2. Génération d'une (ou plusieurs) nouvelle(s) application(s)
 3. Configuration des settings

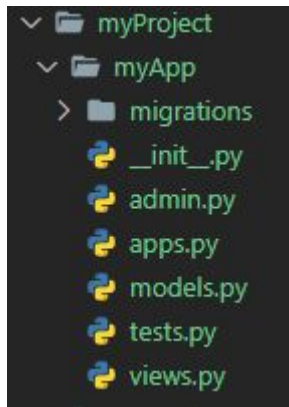
- Les commandes correspondantes sont (dans un environnement virtuel):
 1. `django-admin startproject <project_name>`
 2. `py manage.py startapp <app-name>`
 3. Modification du fichier **settings.py**

Description des fichiers du projet



- **`__init__.py`** □ Fichier vide spécifiant qu'il s'agit d'un paquet
- **`asgi.py` et `wsgi.py`** □ Points d'entrée d'une app Web
- **`settings.py`** □ Fichier de configuration du projet complet
- **`urls.py`** □ Listes des URL's du projet / des app's
- **`manage.py`** □ Outils permettant d'utiliser les commandes Django

Description des fichiers d'une app



- **/migrations** □ Dossier contenant toutes les migrations (auto) de la DB
- **admin.py** □ Fichier d'affichage de la console d'administration
- **apps.py** □ Fichier de configuration de l'application
- **models.py** □ Liste des modèles de l'app (classes)
- **tests.py** □ Liste des TU
- **views.py** □ Liste des vues (pages) de l'application

Configuration et lancement

- Afin de terminer l'initialisation, il faut **configurer** le projet :
 1. Ajouter votre application au projet dans le fichier **settings.py**:
 - Section **INSTALLED_APPS**
 - Ajouter '<app_name>.apps.<App_name>Config' (Exemple: App1 => 'app1.apps.App1Config')
 2. Initialisation de la DB: **py manage.py migrate** (*première migration + fichier db.sqlite3*)
 3. Création d'un superAdmin: **py manage.py createsuperuser** (*à conserver...*)
- Il est alors possible de **lancer** l'application :
py manage.py runserver
- Et de se **connecter** à la console d'administration avec le user crée :
Go to <http://127.0.0.1:8000/admin/>

Exemple

1. Lancement de l'application

```
$ py manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
August 16, 2022 - 11:40:26
Django version 4.1, using settings 'Ynov.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

2. Connexion à l'interface d'administration

Administration de Django

Site d'administration

AUTHENTIFICATION ET AUTORISATION

Groupes

 Ajouter  Modification

Utilisateurs

 Ajouter  Modification

Actions récentes

Mes actions

Aucun(e) disponible

Exercice 1

1. Créer un environnement virtuel
2. Activer le venv
3. Installer la librairie Django
4. Copier (*automatiquement*) les librairies dans un **requirement.txt**
5. Créer le projet « Ynov »
6. Créer l'application « myApp1 » + Ajouter votre config d'app dans les settings
7. Initialiser la base de données SQLite
8. Créer un user administrateur
9. Lancer le serveur, accéder à votre app et s'y connecter



3.1.4. Les modèles



Introduction aux modèles

- **Python** permet de définir des **objets** avec un ensemble d'attributs
- **Django** va avoir la même logique avec des **modèles**
- Un modèle est une **classe** Python
- Un modèle est un **objet** qui va être **stocker** en **base de données**
- Un modèle utilise **l'héritage** d'un module Django: **django.db.models.Model**

Champs de modèle

- Un **champ** de modèle (class **Field**) correspond à **attribut** de classe
- Django fait la **liaison** entre les **modèles** et les **données** de la BD
- Un champs avec Django permet :
 - de définir la nature du champs (chiffre, lettre, date, boolean ...)
 - de créer des relations
 - d'ajouter des options (*valeur par défaut, champs obligatoire, clé primaire ...*)

Type de champs

□ Django propose énormément de **type** de champs, dont les plus classiques :

- Chaîne de caractère: **CharField(), TextField()**
- Nombres: **IntegerField()**
- Nombres flottants: **FloatField()**
- Booléens: **BooleanField()**
- Dates : **.DateField(), DateTimeField(), TimeField()**
- ...

<https://docs.djangoproject.com/fr/4.0/ref/models/fields/#model-field-types>

Type de champs – Relations

- Pour représenter les **relations** entre les **modèles**, Django propose des champs spécifique de relation :

- Un-a-plusieurs:

ForeignKey(<modèle_name>, on_delete, options++)

- Plusieurs-a-Plusieurs:

ManyToManyField(<modèle_name>, options++)

Options de champs

□ Django propose énormément **d'option** de champs, dont les plus classiques :

- Champs nul: **null**
- Champs vide: **blank**
- Valeur par défaut: **défault**
- Clé primaire: **primary_key** (id par défaut)
- Champs unique : **unique**
- ...

□ Chaque type de champs à ses propres **spécificités** et permet des **options supplémentaires** (taille max, liens autre modèle, ...)

Options de champs - Enums

- Une des options possibles est la création d'une **liste de constantes** (Enums Python)

- Il y a deux étapes :
 1. Création de la liste de constante
 2. Ajout de l'option « **choices** » faisant références à la liste de constantes

Mise à jour de la Base de Données

- Chaque modification des modèles (ajout, modification, suppression..) doit se **répercuter** sur la base de données !
- **Deux commandes** doivent donc être réalisées :
 1. `py manage.py makemigrations <app_name>`
 2. `py manage.py migrate`
- Pour **visualiser** les correspondances en base (pour du SQLite) :

<https://sqlitebrowser.org/dl/>

Exemple

```
class Ecole(models.Model):
    nom = models.CharField(max_length=30)
    anneeConstruction = models.CharField(max_length=4, null=True)

class Classe(models.Model):
    NIVEAU = [
        ('LIC', 'License'),
        ('M1', 'Master 1'),
        ('M2', 'Master 2'),
    ]
    code = models.CharField(max_length=30)
    nbEleves = models.IntegerField(default=5)
    niveau = models.CharField(max_length=3, default='LIC', choices=NIVEAU)
    ecole = models.ForeignKey(Ecole, on_delete=models.PROTECT)
```

Exemple

1. Génération migration

```
py manage.py makemigrations exemple
```



```
←[36;1mMigrations for 'exemple':←[0m  
←[1mexemple\migrations\0001_initial.py←[0m  
- Create model Ecole  
- Create model Classe
```

2. Lancement migrations

```
py manage.py migrate
```



```
Applying exemple.0001_initial...←[32;1m OK←[0m
```

3. Vérification en base :

exemple_session		CREATE TABLE "exemple_session" ("session_id" varchar(10) NOT NULL PRIMARY KEY, "session_data" text)
▼ exemple_classe		CREATE TABLE "exemple_classe" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "nbEleves" integer NOT NULL)
id	integer	"id" integer NOT NULL
nbEleves	integer	"nbEleves" integer NOT NULL
niveau	varchar(3)	"niveau" varchar(3) NOT NULL
ecole_id	bigint	"ecole_id" bigint NOT NULL
code	varchar(30)	"code" varchar(30) NOT NULL
▼ exemple_ecole		CREATE TABLE "exemple_ecole" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "anneeConstruction" varchar(4) NOT NULL, "nom" varchar(30) NOT NULL)
id	integer	"id" integer NOT NULL
anneeConstruction	varchar(4)	"anneeConstruction" varchar(4) NOT NULL
nom	varchar(30)	"nom" varchar(30) NOT NULL

Exercice 2

1. Créer 2 modèles (models.py) répondant aux exigences suivantes :
 - Artiste (nom, style avec la liste suivante : [POP, RAP, CLASSIC, ROCK, UNDEFINED])
 - Chanson (titre, durée, date, artiste avec la référence au modèle Artiste)
2. Générer le(s) fichier(s) de migration
3. Générer la migration
4. Ouvrir la base données et vérifier que les nouveaux champs sont créés

Interface de programmation

- Django met à disposition une **interface** pour (**tester**) **requêter** les données :

```
py manage.py shell
```

- Vous avez ensuite la possibilité de **manipuler** vos objets avec la syntaxe suivante :

- **CREATE:** `myVar = <model_name>(<field_name> = <value>)`
- **UPDATE:** `myVar.<field_name> = <new_value>`
- **READ**
 - (Par PK) `myVar = <model_name>.objects.get(pk=<primary_Key>)`
 - (Tous) `myVar = <model_name>.objects.get.all()`
 - (Filtres) `myVar = <model_name>.objects.filter(<field_name> = <value_searched>)`
 - Et plein d'autres (`exclude`, `contains`, `lte`, `gte` ...)

- Pour **mettre à jour/supprimer** la DB, il faut utiliser la méthode `save()` (ou `delete()`)

Site d'administration

- Le **site d'administration** (<http://127.0.0.1:8000/admin/>) permet de visualiser les **applications** et **modèles** associés
- Par défaut, seul les **utilisateurs** et **groupes** sont visibles
- Pour **visualiser** un modèle, il faut modifier le fichier « **admin.py** » :
 - **Import** du modèle correspondant
 - **Référencement** du modèle correspondant
- La syntaxe est la suivante :
`admin.site.register(<model_name>)`

Exemple requêtes (1/2)

```
py manage.py shell
```

Création :

```
>>> from exemple.models import Ecole, Classe
>>> ecole1 = Ecole(nom='CentreFormation1', anneeConstruction='2022')
>>> ecole2 = Ecole()
>>> ecole2.nom = 'CentreFormation2'
>>> ecole2.anneeConstruction = 2005
>>> ecole1.save()
>>> ecole2.save()
```



	id	anneeConstruction	nom
	Filtre	Filtre	Filtre
1	1	2022	CentreFormation1
2	2	2005	CentreFormation2

Accès + Création :

```
>>> from exemple.models import Ecole, Classe
>>> ecoleSearched = Ecole.objects.get(pk=1)
>>> classe1 = Classe(code='COM_01', nbEleves=17, niveau='M1', ecole=ecoleSearched)
>>> classe2 = Classe(code='DROIT_01', niveau='LIC', ecole=ecoleSearched)
>>> classe3 = Classe(code='IT_01', niveau='LIC', ecole=ecoleSearched)
>>> classe1.save()
>>> classe2.save()
>>> classe2.save()
>>> classe3.save()
>>> exit()
```



	id	nbEleves	niveau	ecole_id	code
	Filtre	Filtre	Filtre	Filtre	Filtre
1	1	17	M1	1	COM_01
2	2	5	LIC	1	DROIT_01
3	3	5	LIC	1	IT_01

Exemple requêtes (2/2)

```
py manage.py shell
```

Update :

```
>>> from exemple.models import Ecole, Classe
>>> c = Classe.objects.get(pk=2)
>>> c.nbEleves
5
>>> c.nbEleves = 11
>>> c.nbEleves
11
>>> c.save()
>>> exit()
```



	id	nbEleves	niveau	ecole_id	cod
	Filtre	Filtre	Filtre	Filtre	Filtre
1	1	17	M1	1	COM_01
2	2	11	LIC	1	DROIT_01
3	3	5	LIC	1	IT_01

Accès (+filtres) et manipulation:

```
>>> from exemple.models import Ecole, Classe
>>> classes = Classe.objects.all()
>>> classeFiltred = Classe.objects.filter(niveau='LIC')
>>> len(classes)
3
>>> len(classeFiltred)
2
>>> classes[0].niveau
'M1'
```

Exemple Graphique (site d'admin)

1. Fichier admin.py :

```
from django.contrib import admin
from .models import Classe, Ecole

admin.site.register(Ecole)
admin.site.register(Classe)
```

2. Lancement du server

```
py manage.py runserver
```

3. Go to <http://127.0.0.1:8000/admin/>

EXEMPLE

Classes

+ Ajouter ✎ Modification

Ecoles

+ Ajouter ✎ Modification

4. Effectuer des modifications, ajouts

Exercice 3

1. En lignes de commandes, créer :
 - 2 artistes (1 Pop + 1 Rap)
 - 5 chansons (3 pour l'artiste 1 + 2 pour l'artiste 2)
2. Ajouter les 2 modèles dans le site d'administration
3. Lancer le serveur
4. Visualiser les artistes et chansons créées
5. Ajouter 1 chanson (interface graphique) pour l'artiste 1

Coming soon

Avant de voir la suite, le cours sera complété pour approfondir la partie monolithique de Django (ajout du Front)

3.1.5. Django Rest Framework

Qu'est ce que DRF ?

- **Django Rest Framework** est une **bibliothèque** permettant de mettre en place rapidement une **API**
- L'architecture proposée permet de s'appuyer sur les **modèles** Django
- **Django Rest Framework** va mettre à disposition un ensemble de **routes** permettant de manipuler les **objets** de la base de données
- L'avantage du Framework est sa **rapidité** de mise en place de l'API

Prérequis

□ Pour utiliser **Django Rest Framework**, 3 étapes sont nécessaires :

1. Ajouter la librairie au projet :

`pip install djangorestframework` (dans l'environnement virtuel !)

2. Déclarer DRF dans la liste des applications (`settings.py`) :

INSTALLED_APPS □ Ajouter `'rest_framework'`

3. Activer la méthode d'authentification DRF (`urls.py`) :

urlpatterns □ Ajouter `'path('api-auth/', include('rest_framework.urls'))'`

Endpoints et Sérializers

- Un **Endpoint** est une extrémité d'un canal de communication qui fonctionne avec l'**API**
- Un **serializer** permet de transformer/formater un **modèle** pour l'API (*données JSON par exemple*)
- Il faut créer un fichier **serializers.py** et importer la classe **ModelSerializer** du module **rest_framework.serializers**
- Le serializer définit le **modèle** de donnée correspondant + les **champs**
- Une bonne pratique consiste à nommer le serializer:
<ModelName>Serializer(ModelSerializer)

Les vues (views)

- Les **vues** permettent de réaliser un **traitement** et renvoyer une **réponse** formatée pour l'API
- Toutes les vues sont à définir dans le fichier **views.py**
- Le fichier des vues va définir les **modèles** de vues, les **méthodes**, les **traitements** et les **réponses**
- Il faut importer les classes **APIView** du module **rest_framework.views** et **Response** du module **rest_framework.reponse**
- Une bonne pratique consiste à nommer la vue: **<ModelName>APIView**

Les routes

- Les **routes** permettent de définir les **urls d'accès** aux vues pour l'**API**
- Toutes les routes sont à définir dans le fichier **urls.py**
- Ce fichier va permettre de lister les urls et d'y définir (au minimum) les **CRUD** (*Create, Read, Update, Delete*)
- Une bonne pratique consiste à **nommer** les routes :
 api/<ModelName>s/
 api/<ModelName>s/:id
 ...

Accès à l'API

□ Avec ces différentes étapes, cela permet de **créer une nouvelle route**

□ Il faut alors relancer le serveur et accéder aux URL :

<http://127.0.0.1:8000/api-auth/>

<http://127.0.0.1:8000/api/<ModelName>s/>

<http://127.0.0.1:8000/api/<ModelName>s/15>

...

Exemple (1/2)

1. Ajout de la librairie DRF (*req.txt*)

```
pip install djangorestframework
```

3. Ajout Authentification DRF

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('api-auth/', include('rest_framework.urls'))],
```

2. Update settings.py

```
INSTALLED_APPS = [  
    'rest_framework',  
    'exemple.apps.ExempleConfig',
```

4. Création des serializers

```
class EcoleSerializer(ModelSerializer):  
    class Meta:  
        model = Ecole  
        fields = ['id', 'nom', 'anneeConstruction']
```

5. Création des vues

```
class EcoleAPIView(APIView):  
    def get(self, *args, **kwargs):  
        categories = Ecole.objects.all()  
        serializer = EcoleSerializer(categories, many=True)  
        return Response(serializer.data)
```

6. Ajout de la route

```
path('api/ecoles/', EcoleAPIView.as_view()),
```


Exemple (2/2)

Lancement du serveur

```
$ py manage.py runserver  
Watching for file changes with StatReloader  
Performing system checks...
```

<http://127.0.0.1:8000/api-auth/login/>

<http://127.0.0.1:8000/api/ecoles/>

Ecole Api

OPTIONS

GET

GET /api/ecole/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
[  
  {  
    "id": 1,  
    "nom": "CentreFormation1",  
    "anneeConstruction": "2022"  
  },  
  {  
    "id": 2,  
    "nom": "CentreFormation2",  
    "anneeConstruction": "2005"  
  }  
]
```

Django REST framework

Nom d'utilisateur:

Mot de passe:

Log in

Exercice 4

1. Ajouter le Django Rest Framework au projet (et au requirement.txt !)
2. Intégrer la route d'authentification proposée par DRF
3. Créer un serializer pour le modèle d'Artiste (avec tous les champs)
4. Créer une vue pour récupérer tous les artistes
5. Créer une route pour afficher tous les artistes
6. Lancer le serveur + Tester

Artiste Api

OPTIONS

GET

GET /api/artistes/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[
  {
    "id": 1,
    "nom": "Rico",
    "style": "POP"
  },
  {
    "id": 2,
    "nom": "Verdo",
    "style": "RAP"
  }
]
```



3.1.6. Routes HTTP



Méthodes HTTP

- Les **méthodes HTTP** sont en corrélation avec les méthodes de **manipulation** des données en base.

- Pour toutes les méthodes de **CRUD**, il y a une **méthode HTTP** associée :
 - **CREATE** Création d'une ressource **POST**
 - **READ** Récupération d'une/plusieurs ressources **GET**
 - **UPDATE** Mise à jour d'une/plusieurs ressources **PUT**
 - **DELETE** Suppression d'une/plusieurs ressources **DELETE**

Router

- La mise en place des différentes routes peut vite devenir **illisible**
- DRF propose une **classe** pour mettre en place un **routage** pour définir automatiquement toutes les **URLs** accessibles
- Dans le fichier **urls.py**, il faut:
 1. Importer la classe **routers** du module **rest_framework**
 2. Créer une **variable** pour le routeur
 3. **Déclarer** l'url de base
 4. **Inclure** le routage dans les différentes urls
- Il est également nécessaire de **transformer** les vues en **ModelViewSet** (pour profiter de toutes les méthodes **CRUD**)

Exemple (1/3)

1. Intégration du serveur (*urls.py*)

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers

from exemple.views import EcoleViewSet

router = routers.SimpleRouter()
router.register('ecoles', EcoleViewSet, basename='ecole')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
    path('api/', include(router.urls))
]
```

2. Modification des vues (*views.py*)

```
from rest_framework.viewsets import ModelViewSet
from .models import Ecole
from .serializers import EcoleSerializer

class EcoleViewSet(ModelViewSet):
    serializer_class = EcoleSerializer

    def get_queryset(self):
        return Ecole.objects.all()
```

Exemple (2/3) □ GET ALL + POST

<http://127.0.0.1:8000/api/ecoles/>

Ecole Viewset List

OPTIONS GET

GET /api/ecoles/

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
  {
    "id": 1,
    "nom": "CentreFormation1",
    "anneeConstruction": "2022"
  },
  {
    "id": 2,
    "nom": "CentreFormation2",
    "anneeConstruction": "2005"
  }
]
```

Raw data HTML form

Nom

AnneeConstruction

POST

Exemple (3/3) □ GET + PUT + DELETE

<http://127.0.0.1:8000/api/ecoles/1>

Ecole Viewset Instance

DELETE

OPTIONS

GET ▾

GET /api/ecoles/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
  "id": 1,  
  "nom": "CentreFormation1",  
  "anneeConstruction": "2022"  
}
```

Raw data

HTML form

Nom

CentreFormation1

AnneeConstruction

2022

PUT

Ajout de filtres

- Les vues permettent de **customiser/modifier** la récupération des données à afficher
- Il faut utiliser les **filtres** de Django et modifier la méthode **get_queryset(self)**

```
class EcoleViewSet(ModelViewSet):
    serializer_class = EcoleSerializer

    def get_queryset(self):
        # Récupération de tous les objets
        queryset = Ecole.objects.all()

        # Récupération du paramètre dans l'url
        annee = self.request.GET.get('annee')

        # Filtre s'il y a un parametre
        if annee is not None:
            queryset = queryset.filter(anneeConstruction=annee)
        # Envoie de la réponse (complète ou partielle)
        return queryset
```

<http://127.0.0.1:8000/api/ecoles/>

<http://127.0.0.1:8000/api/ecoles/?annee=2022>

Exercice 5

1. Mettre en place le router dans les urls (Artiste)
2. Ajouter dans les views des ModelAndViewSet (Artiste)
3. Ajouter 2 filtres possibles dans le GET: *nom* ou *style*
4. Lancer le serveur + Tester les 7 méthodes :
 - GET ALL
 - POST
 - GET BY ID
 - PUT
 - DELETE
 - GET (filtré par nom)
 - GET (filtré par style)

3.1.7. Serializers et relations

Relations entre les modèles (1/2)

- La création des modèles impliquent des **relations** entre eux (Clé étrangère)
- Par **défaut**, l'appel d'une route comportant un modèle **avec une clé étrangère** affichera uniquement la **valeur** de cette clé étrangère
- Il faut logiquement ajouter dans le **serializer** le champs à **afficher**

Classe Viewset List

OPTIONS

GET

GET /api/classes/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[  
  {  
    "id": 1,  
    "code": "COM_01",  
    "nbEleves": 17,  
    "niveau": "M1",  
    "ecole": 1  
  },  
  {  
    "id": 2,  
    "code": "DROIT_01",  
    "nbEleves": 11,  
    "niveau": "LIC",  
    "ecole": 1  
  }  
]
```

Relations entre les modèles (2/2)

- DRF propose également d'afficher les **relations inverses (oneToMany)**.
- Pour afficher les données sous forme de liste, il est nécessaire d'ajouter une **référence de nommage** dans le modèle correspondant (paramètre **related_name**)
- Par **défaut**, l'appel d'une route comportant un modèle **avec la relation oneToMany** affichera uniquement la(les) **valeur(s)** des clés primaires

Ecole Viewset List

OPTIONS

GET

GET /api/ecoles/

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

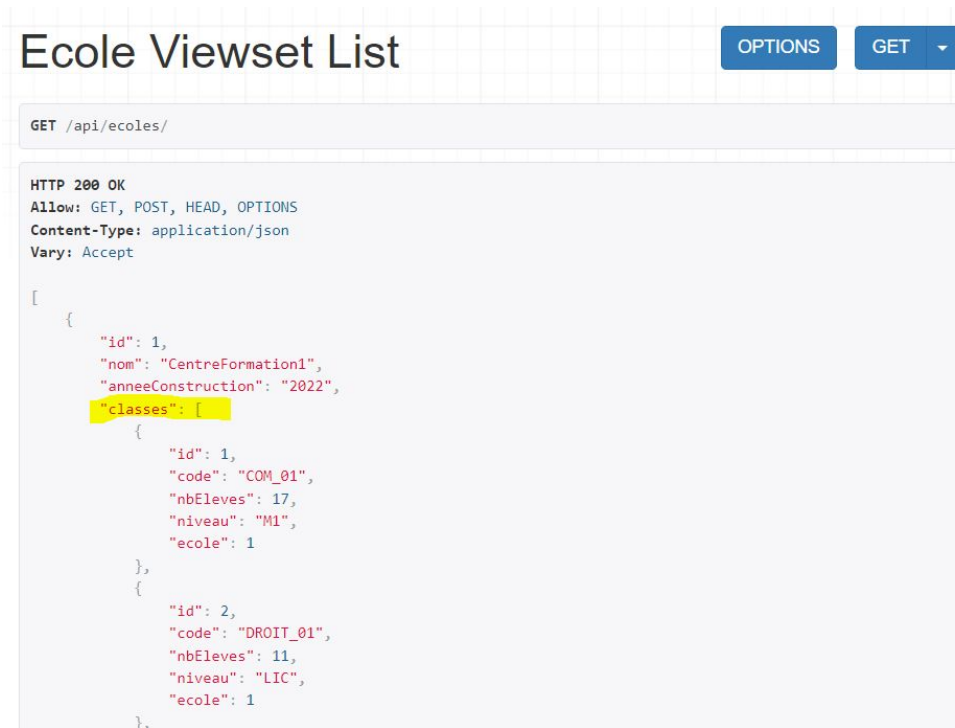
Content-Type: application/json

Vary: Accept

```
[
  {
    "id": 1,
    "nom": "CentreFormation1",
    "anneeConstruction": "2022",
    "classes": [
      1,
      2,
      3
    ]
  },
]
```

Customiser l'affichage

- Pour afficher **toutes les données** d'un modèle (*et non pas l'id uniquement*), il faut appeler le serializer correspondant en y ajoutant l'argument « **many = True** »
- L'option **depth** permet de définir automatiquement le **nombre de niveau de dépendances**



Exemple

1. Ajout de la référence de nommage

```
class Ecole(models.Model):
    nom = models.CharField(max_length=30)
    anneeConstruction = models.CharField(max_length=4, null=True)

class Classe(models.Model):
    NIVEAU = [
        ('LIC', 'License'),
        ('M1', 'Master 1'),
        ('M2', 'Master 2'),
    ]
    code = models.CharField(max_length=30)
    nbEleves = models.IntegerField(default=5)
    niveau = models.CharField(max_length=3, default='LIC', choices=NIVEAU)
    ecole = models.ForeignKey(Ecole, related_name='classes', on_delete=models.PROTECT)
```

2. Intégration des relations

```
class ClasseSerializer(ModelSerializer):
    class Meta:
        model = Classe
        fields = ['id', 'code', 'nbEleves', 'niveau', 'ecole']

class EcoleSerializer(ModelSerializer):
    classes = ClasseSerializer(many=True)

    class Meta:
        model = Ecole
        fields = ['id', 'nom', 'anneeConstruction', 'classes']
```

Customiser les serializers

- Toutes les méthodes CRUD utilisées **font référence** à des méthodes du **ModelSerializer** prédéveloppé par le Framework DRF.
- Comme toute méthode, il est possible de les **surcharger** :
 - **create** (self, validated_data)
 - **update** (self, instance, validated_data):
- Détails :
 - **self** □ Objet courant
 - **validated_data** □ Dictionnaire des données récupérées de la méthode POST
 - **Instance** □ Instance d'objet courant

Customiser les serializers - Exemple

```
class EcoleSerializer(ModelSerializer):  
    # classes = ClasseSerializer(many=True)  
  
    class Meta:  
        model = Ecole  
        fields = ['id', 'nom', 'anneeConstruction', 'classes']  
        depth = 1  
  
    def create(self, validated_data):  
        validated_data['nom'] = "NouveauNom"  
        instance = Ecole.objects.create(**validated_data)  
        return instance  
  
    def update(self, instance, validated_data):  
        # Update a faire ...  
        return instance
```

ViewSet et APIView

- On a vu qu'il était possible d'implémenter des **ViewSet** et des **APIView**, il est possible de **cumuler** ces deux options
- On préférera utiliser les **ViewSet** proposées par DRF car elle mettent à disposition une **logique** et une **rapidité** de mise en œuvre des CRUD
- Pour des besoins **plus spécifiques**, il est toujours intéressant de pouvoir implémenter une route, une logique ou une méthode **personnalisée**,

Exercice 6

1. Ajouter les routes (CRUD) pour les chansons (*serializers, urls, views, ...*)
2. Intégrer l'affichage des chansons pour chaque Artiste

Chanson Viewset List

GET /api/chansons/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[
  {
    "id": 1,
    "titre": "Je vole",
    "duree": 2.56,
    "date": "2022-08-19",
    "artiste": 1
  },
  {
    "id": 2,
    "titre": "Recommence",
    "duree": 2.49,
    "date": "2021-12-20",
    "artiste": 1
  }
]
```

Artiste Viewset Instance

GET /api/artistes/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
  "id": 1,
  "nom": "Rico",
  "style": "POP",
  "chansons": [
    {
      "id": 1,
      "titre": "Je vole",
      "duree": 2.56,
      "date": "2022-08-19",
      "artiste": 1
    },
    {
      "id": 2,
      "titre": "Recommence",
      "duree": 2.49,
      "date": "2021-12-20",
      "artiste": 1
    }
  ]
}
```

Exercice 7

Créer une url
supplémentaire pour lister
tous les noms des
chansons

Chanson Api

GET /api/chansons/display/noms

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "chansons": [
    "Je vole",
    "Recommence",
    "Mes ailes",
    "La villa",
    "Dans mon club",
    "Nous revoilà"
  ]
}
```



3.1.8. Authentification avec JWT

JWT, c'est quoi ?

- L'authentification permet de rendre certains endpoints privés et accessibles seulement aux utilisateurs authentifiés.
- JWT est le sigle de JSON Web Token, qui est un jeton d'identification communiqué entre un serveur et un client.
- Le JWT est utilisé pour s'assurer de l'identité de la personne réalisant la requête. Lorsque le serveur reçoit une requête, il vérifie alors la validité du token et détermine l'utilisateur à l'initiative de la requête. Le JWT permet d'identifier l'utilisateur à l'origine de la requête et permet ainsi de vérifier ses droits.
- Librairie Simple JWT implémente ce système et est préconisée par DRF

Installez et configurez JWT sur Django

- La librairie Simple JWT va nous permettre d'authentifier nos utilisateurs et de leur fournir une paire de JWT :
 - ▶ Un `access_token` qui va permettre de vérifier l'identité et les droits de l'utilisateur. Sa durée de vie est limitée dans le temps.
 - ▶ Un `refresh_token` qui va permettre d'obtenir une nouvelle paire de tokens une fois que l'`access_token` sera expiré.

- Deux endpoints vont donc être mis à disposition par `django-rest-framework-simplejwt` :
 - ▶ Un endpoint d'authentification
 - ▶ Un endpoint de rafraîchissement de token.

Exemple mise en place

1. Ajout de la librairie simplejwt

```
pip install django-rest-framework-simplejwt
```

2. Update settings.py

```
INSTALLED_APPS = [  
    'rest_framework',  
    'rest_framework_simplejwt',
```

3. Ajout des routes

```
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView  
  
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('api-auth/', include('rest_framework.urls')),  
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),  
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
```

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',  
    'PAGE_SIZE': 100,  
    'DEFAULT_AUTHENTICATION_CLASSES': ('rest_framework_simplejwt.authentication.JWTAuthentication',)  
}  
  
SIMPLE_JWT = {  
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),  
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),  
}
```

4. Limitez l'accès aux vues pour les utilisateurs authentifiés

```
from rest_framework.permissions import IsAuthenticated  
  
class EcoleViewSet(ModelViewSet):  
    serializer_class = EcoleSerializer  
  
    permission_classes = [IsAuthenticated]
```


Exemple récupération token

Token Obtain Pair

Takes a set of user credentials and returns an access and refresh JSON web token pair to prove the authentication of those credentials.

POST /api/token/

HTTP 200 OK

Allow: POST, OPTIONS

Content-Type: application/json

Vary: Accept

```
"refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t1b190eXB1IjoicmVmcmVzaSI6ImV4Ci6MTY3NDg5NzU1MCwianRpIjoiOTJjZTVlYTl0MDAxNGMzMjk3NDdIZGUyZWY4ZDY0STkiLCJ1c2VyX2lkIjoy  
"access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t1b190eXB1IjoieWYnZlZXNzIiwiaWF0IjoxNDkxODExNDUwLjQwdGkiOiIyMGNNKjQzOTZTNkZke0YlUTY7Y2kyZQwMDI4NzU0GEicXI6InVnZlZ3fawQiojJ9.
```

Raw data

HTML form

Username

Password

POST

Exemple rafraîchissement token

Token Refresh

[OPTIONS](#)

Takes a refresh type JSON web token and returns an access type JSON web token if the refresh token is valid.

POST /api/token/refresh/

HTTP 200 OK

Allow: POST, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t1b190eXB1IjoiaWVhbnR5c000eXNDg0LCJqdGkiOiJhYjYjc0Y2M3NTkzYjA0ZjU2YmVkNzkyZWY0WY0MjBkZiIsInVzZXJfawQiOiJ9."
}
```

[Raw data](#)[HTML form](#)

Refresh

POST

Postman, LE client pour tester nos API

□ Qu'est-ce qu'un client ?

□ Postman :

- ▶ Plus complet
- ▶ Idéal pour développer des APIs
- ▶ Gestion de l'auth
- ▶ Code snippet
- ▶ Et plus encore !



POSTMAN

□ <https://www.postman.com/downloads/>

Exemple (1/3)

The screenshot shows a REST client interface with a dark theme. At the top, the URL `http://127.0.0.1:8000/api/classes/` is entered. The method is set to `GET`. Below the URL bar, tabs for `Params`, `Authorization`, `Headers (6)`, `Body`, `Pre-request Script`, `Tests`, and `Settings` are visible. The `Body` tab is selected, showing a JSON response: `{"detail": "Informations d'authentification non fournies."}`. The status bar at the bottom indicates `Status: 401 Unauthorized`, `Time: 10 ms`, and `Size: 435 B`. A `Save Response` button is also present.

`http://127.0.0.1:8000/api/classes/` Save

GET `http://127.0.0.1:8000/api/classes/` Send

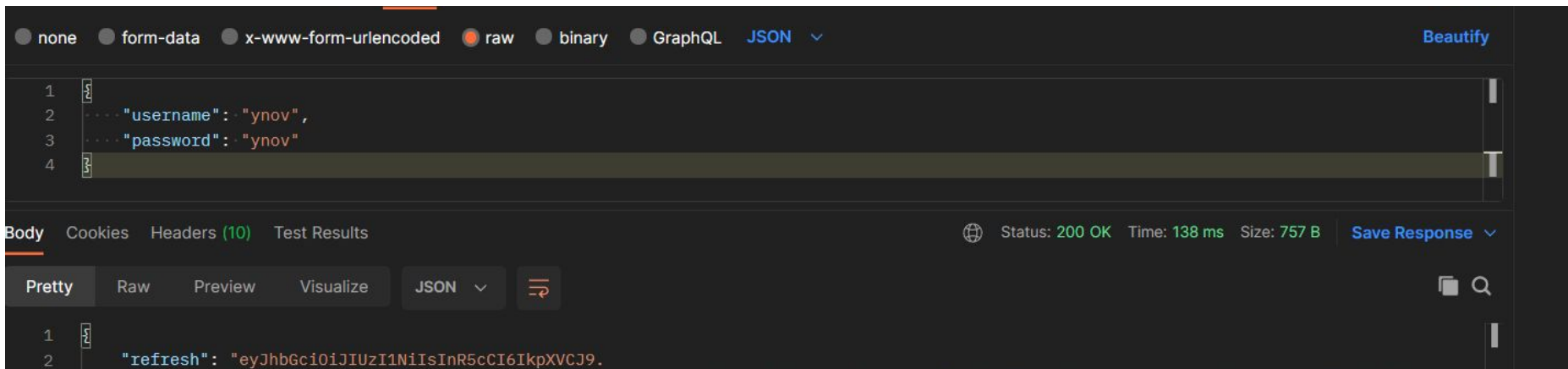
Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (11) Test Results Status: 401 Unauthorized Time: 10 ms Size: 435 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "detail": "Informations d'authentification non fournies."
3 }
```


Example (2/3)




The screenshot shows a web browser's developer tools interface. At the top, there are radio buttons for different content types: none, form-data, x-www-form-urlencoded, raw (selected), binary, GraphQL, and JSON. A 'Beautify' button is on the right. The main area displays a JSON object with two properties: 'username' and 'password', both with the value 'ynov'. Below this, the 'Body' tab is active, showing a status of 200 OK, a time of 138 ms, and a size of 757 B. A 'Save Response' button is also present. The 'Pretty' tab is selected, showing the JSON response: {"refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9."}.

none form-data x-www-form-urlencoded raw binary GraphQL JSON [Beautify](#)

```
1 {  
2   "username": "ynov",  
3   "password": "ynov"  
4 }
```

Body Cookies Headers (10) Test Results [Save Response](#)  Status: 200 OK Time: 138 ms Size: 757 B

Pretty Raw Preview Visualize JSON 

```
1 {  
2   "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9."
```

Example (3/3)

The screenshot shows a REST client interface with the following components:

- Top Tab Bar:** Params, **Authorization** (selected), Headers (7), Body, Pre-request Script, Tests, Settings, and Cookies.
- Authorization Tab Content:**
 - Type:** A dropdown menu showing "Bearer Token".
 - Description:** "The authorization header will be automatically generated when you send the request. [Learn more about authorization](#) ↗"
 - Warning:** "Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. [variables](#) ↗"
 - Token:** A text field containing the token "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ..."
- Bottom Tab Bar:** Body, Cookies, Headers (10), and Test Results.
- Bottom Status Bar:** Includes a globe icon, "Status: 200 OK", "Time: 19 ms", "Size: 749 B", and a "Save Response" button with a dropdown arrow.
- Bottom Footer:** Includes tabs for "Pretty" (selected), "Raw", "Preview", and "Visualize", followed by a "JSON" dropdown and a refresh icon.

Remarques

- Ceci est une implémentation de base de JWT mais de nombreuses choses supplémentaires sont possibles.

Exemples :

- ▶ Gestion des permissions plus fines
 - ▶ Récupérer les infos de l'utilisateur ou autre donné encodé dans le jeton
 - ▶ Divers paramètres
 - ▶ Etc...
-
- Un JWT est constitué de 3 parties séparées par un point. Chaque partie est encodée en base64.
 - ▶ Le header, qui est en général constitué de deux attributs, indique le type de token et l'algorithme de chiffrement utilisé.
 - ▶ Le payload contient les informations utiles que nous souhaitons faire transiter entre le serveur et le client.
 - ▶ La signature est un élément de sécurité permettant de vérifier que les données n'ont pas été modifiées entre les échanges client-serveur. La clé permettant la génération de cette signature est stockée sur le serveur qui fournit le JWT (elle est basée sur la SECRET_KEY de Django).

Exercice 8

Mettre en place l'authentification avec JWT pour toutes les routes et testez votre application avec Postman.

TP Meubles

Créer un projet Django de gestion des stocks de meuble avec une base de données et mettre à disposition une api en se basant sur les spécificités ci-dessous:

- Un Meuble à un **nom**, un **état** (NEUF, OCCASION, MAUVAIS ETAT, INUTILISABLE), un **lieu** de stockage (Magasin), un **prix** et une **statut** (LIBRE, VENDU)
- Un Magasin à un **nom**, une **adresse**, un **Dirigeant**, un **CA** (Chiffre d'affaire)
- Un Dirigeant à un **nom** et un **prenom**

La société « *MeublesDu31* » souhaite avoir une api qui permette au minimum, après authentification :

- De lister tous les meubles
- D'ajouter un meuble (Dans son magasin) □ NEUF par défaut
- De supprimer un meuble
- De changer le statut du meuble
- De vendre un meuble (statut a VENDU, CA du magasin + prix du meuble)

Vous devez utiliser un **environnement virtuel**, une base de données **SQLite** et un fichier **requirements.txt** doit être présent et complet pour lancer le projet.

TP Meubles : Pour aller plus loin

Pour aller plus loin, à l'aide des ressources de votre choix vous mettrez en place les éléments suivants :

Améliorer la sécurité :

- ▶ Mettre en place l'authentification avec JWT
- ▶ Se renseigner sur le CORS pour que les requêtes soient possible uniquement pour des URL données

Tests :

- ▶ Écrivez des tests pour votre API

Proposer des éléments supplémentaires qu'ils vous semble pertinent.



Coming soon

