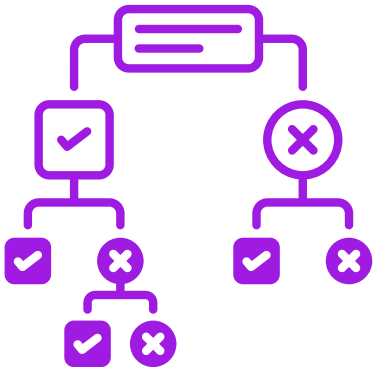
Akinator-1

un jeu qui apprend à deviner

Ce programme est un petit jeu de devinettes inspiré d'Akinator, dans lequel l'ordinateur tente de deviner à quoi pense l'utilisateur — un animal, un objet, une personne, etc.
Mais attention : au départ, l'ordinateur ne connaît presque rien ! Il ne sait poser qu'une seule question, et ne connaît qu'une seule réponse.

Comment ça fonctionne ?
Le jeu se base sur un arbre de décision :



Chaque nœud de l'arbre contient une question (par exemple : "Est-ce un animal ?")
Les feuilles de l'arbre représentent des réponses finales (par exemple : "un chat")
À chaque partie, le programme pose des questions "oui/non" pour naviguer dans l'arbre, jusqu'à arriver à une réponse qu'il propose à l'utilisateur.

Et s'il se trompe ?
C'est là que le jeu devient intelligent !
Quand l'ordinateur se trompe, il demande :

- La bonne réponse à l'utilisateur.
- Une nouvelle question pour distinguer cette réponse de celle qu'il avait proposée.
- S'il faut répondre "oui" ou "non" à cette nouvelle question pour reconnaître la bonne réponse.

Avec ces éléments, il ajoute une nouvelle branche à son arbre. Ainsi, il devient un peu plus intelligent à chaque erreur.
Ce mécanisme simple est un exemple de ce qu'on appelle en informatique l'apprentissage automatique (**machine learning**) :
un programme s'améliore avec l'expérience, en utilisant les nouvelles données qu'on lui donne pour mieux répondre à l'avenir.
Au début, il n'y a qu'une seule question dans l'arbre. Mais plus on joue, plus l'arbre grandit : le nombre de questions augmente, les réponses deviennent plus variées, et l'ordinateur devient de plus en plus performant pour deviner ce que vous avez en tête.

Implémentation en python

Ce projet est complexe et nécessite la maîtrise de 3 notions fondamentale en python:

- La programmation orientée objet
- La structure de données en arbre
- Le format de stockage de données JSON

Nous allons ici faire un bref rappel de ces 3 notions.

La programmation orientée objet

La POO est un paradigme de programmation qui organise le code en objets.
Un objet est une instance d'une classe.
Une classe est comme un plan (un modèle) qui décrit quelles données (attributs) et quelles actions (méthodes) un objet peut avoir.

Classe : définition d'un type d'objet.
Objet : instance d'une classe.
Attributs : variables attachées à un objet (ex : question, réponse).
Méthodes : fonctions attachées à un objet (ex : vérifier si un noeud est une feuille).
Encapsulation : cacher la complexité et protéger les données.
Constructeur : méthode spéciale __init__() pour créer un objet avec des valeurs initiales.

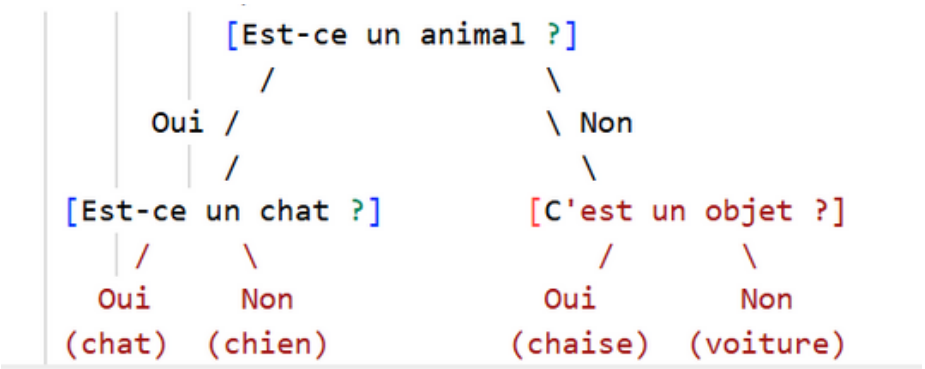
```python
class Node:
    def __init__(self, question=None, answer=None):
        self.question = question
        self.answer = answer
        self.yes = None
        self.no = None

    def is_leaf(self):
        return self.answer is not None
```

Ici, Node est une classe qui représente un élément (un noeud) dans l'arbre du jeu.
Chaque node peut avoir une question, ou une réponse finale (feuille).

Méthode is_leaf() permet de savoir si ce node est une feuille (une réponse).

Les arbres (structure de données)

Un arbre est une structure de données composée de nœuds reliés entre eux par des liens (arêtes).
Le premier nœud s'appelle la racine.
Chaque nœud peut avoir plusieurs nœuds enfants (dans notre cas : 2, oui/non).
Un nœud sans enfants est appelé une feuille.
Pourquoi utiliser un arbre dans Akinator ?
Pour poser des questions en suivant un chemin selon les réponses.
Chaque question correspond à un nœud interne.
Chaque réponse finale (objet deviné) est une feuille.



Ici chaque Node contient 2 références : oui et non vers d'autres nœuds.
Le parcours de l'arbre se fait selon la réponse utilisateur.

JSON (JavaScript Object Notation)

JSON est un format léger pour stocker et échanger des données.
C'est une chaîne de caractères qui représente des objets, listes, valeurs simples.
Facilement lisible par l'homme et manipulable par les langages de programmation.
Pourquoi JSON dans Akinator ?
Pour sauvegarder l'état de l'arbre (questions et réponses) dans un fichier.
Pour pouvoir le recharger à la prochaine utilisation, et que le jeu "se souvienne".

```
{
  "question": "Est-ce un animal ?",
  "yes": {
    "question": "Est-ce un chat ?",
    "yes": {"answer": "chat"},
    "no": {"answer": "chien"}
  },
  "no": {
    "question": "Est-ce un objet ?",
    "yes": {"answer": "chaise"},
    "no": {"answer": "voiture"}
  }
}
```

En Python, pour sauvegarder, on convertit l'arbre en dictionnaire Python, puis on utilise json.dump().
Pour charger, on lit le fichier JSON, on convertit en dictionnaire avec json.load(), puis on reconstitue les objets Node.

Analyse du programme bloc par bloc

```
import json
import os
```

json sert à manipuler des fichiers JSON (pour sauvegarder et charger la base de données du jeu).
os permet de vérifier l'existence d'un fichier (utile pour savoir si la base de données existe déjà).

```
# Classe représentant un noeud dans l'arbre de décision
class Node:
    def __init__(self, question=None, answer=None):
        self.question = question  # Question à poser si ce n'est pas une feuille
        self.answer = answer      # Réponse (seulement pour les feuilles)
        self.yes = None           # Branche si la réponse est "oui"
        self.no = None            # Branche si la réponse est "non"

    # Méthode pour savoir si ce noeud est une feuille (réponse finale)
    def is_leaf(self):
        return self.answer is not None
```

La classe Node représente un noeud de l'arbre.
Chaque Node peut contenir :

- une question (pour nœud interne),
- ou une réponse (pour feuille, c'est-à-dire la proposition finale).
- Les attributs yes et no pointent vers d'autres Node selon la réponse.

La méthode is_leaf() retourne True si le nœud est une feuille (possède une réponse).

```
# Fonction pour poser une question oui/non à l'utilisateur
def ask_yes_no(question):
    while True:
        ans = input(question + " (oui/non) ").lower()
        if ans in ['oui', 'non']:
            return ans == 'oui'  # Retourne True si "oui", sinon False
```

Pose une question à l'utilisateur, attend une réponse "oui" ou "non".
Repose la question tant qu'une réponse valide n'est pas donnée.
Retourne True si la réponse est "oui", sinon False.

```python
# Fonction principale pour jouer avec un noeud donné
def play(node):
    if node.is_leaf():
        # On est sur une feuille : proposer une réponse
        if ask_yes_no(f"Est-ce que tu penses à {node.answer} ?"):
            print("Super ! J'ai deviné ! 😎")
        else:
            # Échec : demander à l'utilisateur la bonne réponse et une question pour la différencier
            print("Zut ! J'ai échoué. 😳")
            correct_answer = input("À quoi pensais-tu ? ")
            new_question = input(f"Donne-moi une question pour distinguer {correct_answer} de {node.answer} : ")

            # Demander si la réponse au nouvel objet est "oui"
            if ask_yes_no(f"Pour {correct_answer}, la réponse à cette question est-elle 'oui' ?"):
                node.question = new_question
                node.yes = Node(answer=correct_answer)
                node.no = Node(answer=node.answer)
            else:
                node.question = new_question
                node.no = Node(answer=correct_answer)
                node.yes = Node(answer=node.answer)

            # Ce noeud n'est plus une feuille
            node.answer = None
    else:
        # Si c'est une question, continuer à descendre dans l'arbre
        if ask_yes_no(node.question):
            play(node.yes)
        else:
            play(node.no)
```

Si le nœud est une feuille :

- Le programme propose sa réponse.
- Si c'est vrai, il affiche un message de succès.
- Sinon, il demande à l'utilisateur la bonne réponse et une question pour la différencier.
- Puis il modifie le nœud actuel pour devenir un nœud interne avec cette nouvelle question, créant deux branches (oui/non).

Sinon (nœud interne) :

- Le programme pose la question du nœud.
- Selon la réponse "oui" ou "non", il appelle récursivement play sur le nœud enfant correspondant.

```python
# Sauvegarder l'arbre dans un fichier JSON
def save_tree(node, filename):
    with open(filename, 'w') as f:
        json.dump(serialize(node), f)

# Charger l'arbre depuis un fichier JSON (ou créer un arbre par défaut)
def load_tree(filename):
    if os.path.exists(filename):
        with open(filename, 'r') as f:
            return deserialize(json.load(f))
    else:
        # Arbre par défaut avec une seule réponse
        return Node(answer="un chat")
```

save_tree : convertit l'arbre en dictionnaire (via serialize) et l'enregistre dans un fichier JSON.
load_tree : charge le fichier JSON s'il existe, le transforme en dictionnaire, puis en arbre avec deserialize. Sinon, crée un arbre par défaut avec une seule réponse "un chat".

```python
# Convertit un arbre en dictionnaire (pour le JSON)
def serialize(node):
    if node is None:
        return None
    if node.is_leaf():
        return {"answer": node.answer}
    return {
        "question": node.question,
        "yes": serialize(node.yes),
        "no": serialize(node.no)
    }

# Reconstruit l'arbre depuis un dictionnaire (après lecture JSON)
def deserialize(data):
    if "answer" in data:
        return Node(answer=data["answer"])
    node = Node(question=data["question"])
    node.yes = deserialize(data["yes"])
    node.no = deserialize(data["no"])
    return node
```

serialize : transforme un arbre de Node en dictionnaire récursivement, prêt à être converti en JSON.
deserialize : fait l'inverse, transforme un dictionnaire JSON en arbre Node.

```python
# === Jeu principal ===

# Nom du fichier contenant la base de connaissances
tree_file = "base_jeu.json"

# Charger l'arbre existant ou en créer un nouveau
root = load_tree(tree_file)

print("Pense à un objet, un animal ou une personne. Je vais deviner ! 🤖")

# Boucle principale du jeu
while True:
    play(root)
    if not ask_yes_no("Veux-tu rejouer ?"):
        break

# Sauvegarde de l'arbre après le jeu
save_tree(root, tree_file)
print("Merci d'avoir joué ! 🌟")
```

Charge l'arbre de décision.
Explique le principe au joueur.
Boucle infinie qui joue la partie, puis demande si on veut rejouer.
À la sortie, sauvegarde la base dans un fichier JSON.
Affiche un message de remerciement.