

Rapport de Soutenance : Projet Black Jack de ATSE Sombo et BAH Mamadou

1. La Structure de Données de Chaque Élément du Jeu

Notre jeu repose sur des structures de données Python claires et efficaces (dictionnaires et liste) pour gérer la complexité du Black Jack.

A. Le Sabot (Deck)

- **Notre Structure** : Une liste (liste) contenant des dictionnaires (dict) pour chaque carte.
- **Notre Contenu** : Le sabot est initialisé à **six jeux de cartes** complets par défaut, soit environ 312 cartes.
- **Notre Initialisation** : La fonction `charger_sabot()` essaie de lire la structure de base des cartes depuis un fichier JSON (`cartes.json`) (Ou sinon crée le fichier directement), la multiplie, puis la mélange aléatoirement (`random.shuffle`).
- **Notre Gestion** : Nous avons mis en place une vérification dans `piocher_carte()` qui déclenche automatiquement le remélange du sabot dès qu'il descend sous 52 cartes, assurant ainsi une continuité de jeu fluide.

Élément	Type	Clés principales
Carte	Dictionnaire	"numero", "symbole", "valeur"

B. La Table de Jeu (table)

- **Notre Structure** : Le dictionnaire principal qui centralise l'état de la partie.
- **Notre Contenu** : Elle contient la banque, le dictionnaire de tous les joueurs (humains et robots) et l'état des places.

Élément	Type	Rôle
"banque"	Dictionnaire	Contient la main et le score de la banque.
"joueurs"	Dictionnaire	Notre dictionnaire de joueurs, où chaque clé est un nom de joueur/robot.
"places"	Liste	C'est notre liste ordonnée des joueurs, essentielle pour la gestion des tours de jeu et l'affichage.

C. Le Joueur / Robot (Contenu de "joueurs")

- **Notre Structure** : Chaque participant est un dictionnaire.

Clé	Type	Notre Description
"main"	Liste	La liste des cartes actuelles du joueur/robot.
"score"	Integer	Le score actuel, calculé par notre fonction <code>score()</code> .
"solde"	Integer	L'argent disponible pour le joueur.

"mise"	Integer	Le montant misé sur la main principale pour la manche.
"assurance"	Integer	Le montant misé pour l'assurance (si choisie, sinon 0).
"statut"	Boolean	Indique si le joueur est actif, a Bust ou est en attente de la prochaine manche.
"est_robot"	Boolean	Indique si nous avons affaire à un joueur automatisé.
"strategie"	Strings	La stratégie choisie pour le robot ('P', 'N', 'A') ou None pour un humain.

2. Nos Commandes de Jeu et Menus

Nous avons conçu un système de saisie en ligne de commande, où toutes les entrées sont validées par nos fonctions `veri_saisie_liste()` et `veri_saisie_intervale()`.

_Démarrage : (O)uvrir une partie sauvegardée / (N)ouvelle partie.

_Ajout de Joueur (`gerer_nouveaux_joueurs`) :

- Ajouter un joueur ? (O/N)
- Est-ce un robot ? (O/N)
- Si robot, choix de la stratégie : (P)rudent, (N)ormal, (A)gressif.
- Choix de la place : L'utilisateur choisit un numéro de place ([0, 1, 2...]).

_Tour de Jeu (Humain) (`tours_joueurs`) :

- Les options sont affichées dynamiquement : ['T', 'R'].
- Si les conditions sont remplies (2 cartes, solde suffisant) : ['T', 'R', 'D'] (Doubler).
- Si les 2 cartes sont identiques : ['T', 'R', 'S'] (Split) (ou ['T', 'R', 'D', 'S'] si les deux sont vrais).

_Actions du Joueur (Tour de Jeu) :

Commande	Notre Rôle	Notre Condition
T	Tirer (Hit) : Demander une carte.	Toujours disponible.
R	Rester (Stand) : Conserver le score.	Toujours disponible.
D	Doubler (Double Down) : Doubler la mise pour une unique carte.	Seulement si la main contient 2 cartes ET le solde est suffisant.
S	Split : Séparer la main en deux mains distinctes.	Seulement si 2 cartes de même valeur ET le solde est suffisant.
O/N	Assurance : Miser sur un Blackjack de la banque.	Seulement si la banque a un AS visible ET le solde est suffisant.

_Aide au Joueur (Conseils) :

Dans la fonction `tours_joueurs`, avant de demander l'action de l'humain, le code analyse la situation :

1. Conseil de Doubler (Prioritaire) :

- `if peut_doubler and 9 <= joueur['score'] <= 11:`
- Le programme vérifie si l'action "Doubler" est possible et si le score est de 9, 10, ou 11 (le meilleur moment pour doubler). Si oui, il affiche "Nous vous conseillons de Doubler".

2. Conseil de Tirer/Rester (Standard) :

- `else: print(f"Nous vous conseillons de {'Tirer' if joueur['score'] < 17 else 'Rester'}...")`
- Si le conseil de doubler n'est pas pertinent, le script applique la règle de la banque (tirer sous 17, rester à 17+) comme conseil de base.

3. Conseil de Splitter (Additionnel) :

- `if peut_splitter and 8 <= joueur['score'] <= 16:`
- En plus du conseil de base, le jeu vérifie si un "Split" est possible et si le score est pertinent (ex: une paire de 8, qui vaut 16, est un excellent "Split"). Il affiche alors "Nous vous conseillons aussi de spliter".

_Fin de Manche (Black_Jack) :

- (S)auvegarder et Quitter : Sérialise la table et le sabot en JSON.
- (Q)uitter : Écrase le JSON de sauvegarde avec `{}` pour le "supprimer".
- (C)ontinuer : Relance la boucle de jeu.

Points Forts et Particularités :

_Robustesse des Fichiers : Auto-création : `charger_sabot()` utilise un `try...except`. Si `cartes.json` est manquant ou corrompu (erreur `FileNotFoundError` ou `json.JSONDecodeError`), la fonction le crée et le remplit automatiquement avant de continuer.

_Robustesse de la Saisie : Notre fonction `veri_saisie_liste` gère les erreurs de saisie (limite de 3 essais).

Notre fonction `add_player` gère les échecs de saisie de l'utilisateur : si un joueur (humain ou robot) ne choisit pas de place, le programme lui en assigne une aléatoirement au lieu d'annuler son entrée.

_Re-mélange Automatique : La fonction `piocher_carte()` vérifie la taille du sabot avant chaque pioche. S'il reste moins de 52 cartes, elle appelle `charger_sabot()` pour re-mélanger 6 nouveaux jeux, garantissant que le jeu ne tombe jamais à court de cartes.

_Gestion de la Partie Plurimanche et Multijoueur : Notre structure permet un jeu continu sur plusieurs manches, gérant la présence simultanée de joueurs humains et de robots. Nous avons aussi prévu la réintégration des joueurs en attente (statut=False) à chaque nouvelle manche.

_Implémentation Complète des Règles : Nous avons codé les règles avancées du Black Jack, incluant : le gain de **3:2** pour le Black Jack, la gestion de l'**Assurance**, le **Doubler (Double Down)**, et le **Séparer (Split)** (avec la gestion des deux mains qui en découlent).

_Résolution du Problème de l'As : Notre fonction score() gère de manière robuste l'As (1 ou 11) en le dévaluant de 11 à 1 si le score total dépasse 21, assurant ainsi le meilleur score possible.

_Diversité des Adversaires (Robots) : Nous offrons trois stratégies de robots différentes, ce qui enrichit l'expérience de jeu et permet aux utilisateurs de faire face à des adversaires aux comportements variés.

_Fonction de Sauvegarde/Chargement : Nous utilisons json pour sauvegarder l'état complet de la partie – y compris l'ordre des cartes restantes dans le sabot – permettant une reprise immédiate et fidèle du jeu.

_Sécurité des Entrées Utilisateur : Nous protégeons le programme contre les erreurs de saisie grâce à nos fonctions de vérification. De plus, nous avons implémenté un **quota d'erreur** qui éjecte un joueur après trop de tentatives invalides, renforçant la stabilité du jeu.

_Améliorations Visuelles et Conclusion : Pour améliorer la jouabilité, nous avons utilisé la bibliothèque colorama :

1. **Couleur des Résultats :** Dans déterminer_resultats(), les messages de gain ("BLACKJACK", "Vous gagnez...") sont affichés en vert, et les messages de perte ("BUST", "Vous perdez...") sont en rouge. Tandis que ceux d'égalités seront en bleu
2. **Titres :** Les titres de section ("NOUVELLE MANCHE", "TOUR DE LA BANQUE") sont affichés en jaune ou autre pour une meilleure lisibilité.

Conclusion

Ce projet nous a permis de mettre en pratique la gestion de structures de données complexes (dictionnaires imbriqués) sans l'aide des classes. La logique principale est fonctionnelle et respecte les règles du Black Jack. Les ajouts des stratégies de robots et de la sauvegarde démontrent la flexibilité du code.