TI101 – Programmation en Python

FORT BOYARD SIMULATOR

Ce projet vous permettra de travailler sur la création de jeux interactifs en Python, en manipulant des concepts tels que les fonctions, les conditions, les boucles, et l'utilisation des fichiers JSON pour gérer les données externes. Vous devrez également concevoir une interface simple pour que l'utilisateur puisse interagir avec le jeu.

Dépôt intermédiaire : Le samedi 21/12/2024, à midi sur moodle

Dépôt final : Le dimanche 05/01/2025, à midi sur moodle

Soutenance : La semaine du 06/01/2025

Cherifa BEN KHELIL

EFREI – P1, P1-BN, P1-BDX, P1-Plus, P1-INT

Fort Boyard Simulator : Défis, Épreuves et Trésor à la Clé

Dans le cadre du module TI101, vous serez amené à développer un simulateur inspiré du célèbre jeu télévisé **Fort Boyard**. L'objectif est de recréer une expérience où une équipe de joueurs doit réussir plusieurs épreuves et récolter trois clés. Une fois ces clés obtenues, l'équipe pourra participer à l'épreuve finale et tenter de débloquer la salle du trésor.

Objectifs du projet

1. **Création d'une équipe** : Permettre à l'utilisateur du jeu de constituer une équipe de **1 à 3 joueurs**.
2. **Types d'épreuves à implémenter** : Le programme doit proposer les 4 types d'épreuves suivants :
    - **Mathématiques** (ex. : calculs de factorielles, équations)
    - **Hasard** (ex. : jeu de bonneteau, lancer de dés)
    - **Logique** (ex. : jeu de NIM, Tic-Tac-Toe)
    - **Énigme du Père Fouras**

    Pour chaque type, un nombre minimum d'épreuves est requis :
    - Pour les **Mathématiques**, il faut inclure au moins **3 épreuves**.
    - Pour les épreuves de **Hasard**, **2 épreuves** doivent être proposées.
    - Enfin, **1** épreuve de **Logique** est requise.
3. **Gains de clés** : Lors de chaque épreuve, un joueur est sélectionné pour y participer. S'il réussit, il gagne une clé. L'équipe peut accéder à la salle du trésor lorsqu'elle a récolté **3 clés**.
4. **Sauvegarde et Historique des performances (optionnel - bonus)** L'historique des performances des joueurs est **sauvegardé** dans un fichier texte. Ce fichier permettra de consulter les résultats à la fin des épreuves.

Utilisation de fichiers JSON

Dans ce projet, vous aurez l'opportunité d'utiliser des fichiers JSON pour manipuler des données. Ces fichiers contiennent des informations réelles sur les énigmes du **Père Fouras** et les indices nécessaires pour accéder à la **salle du trésor**, provenant du site *Fan-FortBoyard.fr*.

Optimisation et Personnalisation du Code

Nous encourageons les étudiants à :

- **Décomposer** certaines parties du code pour en améliorer l'organisation.
- Modifier ou étendre le programme selon les besoins et justifier chaque modification de manière détaillée.
- Proposer des **fonctions supplémentaires** pour enrichir l'application.

Table des matières

1. Les bibliothèques et fonctions natives autorisées

L'utilisation des modules et fonctions prédéfinies est strictement limitée à ceux spécifiquement autorisés et listés ci-dessous. Il est interdit d'utiliser d'autres fonctions prédéfinies, à l'exception des cas suivants :

- ***Gestion des listes :***
  - Ajout d'éléments avec `append()`.
  - Insertion d'éléments avec `insert()`.
  - Suppression d'éléments à l'aide de `del` (applicable également aux collections).
- ***Opérations de conversion de type (cast) :***
  - Conversion explicite entre types (ex. `int()`, `str()`, etc.).
- ***Manipulation de fichiers :***
  - Utilisation de la fonction `open()` pour lire ou écrire dans des fichiers.
  - Fonctions spécifiques de lecture et écriture dans un fichier (par exemple, `write()`, `read()`, `readlines()`, etc.).
- **Affichage des messages :**
  - Utilisation de la fonction `format()` ou des `f-strings`.

1.1. La bibliothèque random

Le module **`random`** en Python offre des fonctions pour générer des nombres aléatoires et réaliser des opérations aléatoires. Dans ce projet, vous aurez l'occasion d'utiliser deux fonctions :

a) La fonction random.randint(a, b) : génère un nombre entier aléatoire compris entre a et b, incluant les deux extrémités. Cela signifie que si vous appelez `random.randint(1, 6)`, cela renverra un entier aléatoire entre 1 et 6.

Exemple :

```
import random
random.seed(3)  # Fixe la graine pour rendre les résultats reproductibles.
print(random.randint(1, 6))  # Affiche un entier aléatoire entre 1 et 6 (ici, toujours 2 avec la graine fixée).
```

Sortie :

```
2
```

b) La fonction random.choice(sequence) : sélectionne aléatoirement un élément d'une séquence (comme une liste, un tuple ou une chaîne). Par exemple, si vous avez une liste ***noms***, utiliser `random.choice(noms)` renverra un nom au hasard dans cette liste. Cela est utile dans des jeux ou des simulations où un choix aléatoire est requis.

Exemple :

```
import random
noms = ['Alice', 'Bob', 'Charlie']  # Liste des noms
choix_aleatoire = random.choice(noms)  # Choisit un nom aléatoire parmi la liste
print(choix_aleatoire)  # Affiche 'Alice' , 'Bob' ou 'Charlie' au hasard
```

1.2. La bibliothèque json

Le format **JSON** (**J**ava**S**cript **O**bject **N**otation) est un format léger et lisible par l'homme, principalement utilisé pour échanger des données entre un serveur et une application web. Il permet de structurer des informations sous forme de paires clé-valeur, ce qui est similaire à un dictionnaire en Python.

Voici un exemple de fichier JSON nommé **data.json** :

```
[
  {
    "nom": "Pierre",
    "age": 30,
    "ville": "Paris",
    "interets": ["lecture", "sport", "musique"]
  },
  {
    "nom": "Marie",
    "age": 25,
    "ville": "Lyon",
    "interets": ["cuisine", "voyages", "photographie"]
  }
]
```

Ce fichier est constitué d'une liste (délimitée par des crochets []) contenant deux éléments (chacun délimité par des accolades {} et séparé par une virgule). Chaque élément correspond à des informations sur une personne, organisées sous forme de paires clé-valeur pour les attributs ***nom***, ***age***, ***ville*** et ***interets***.

Pour travailler avec un fichier JSON en Python, vous pouvez utiliser la bibliothèque **`json`**. La fonction **`json.load()`** permet de charger les données d'un fichier JSON et de les convertir en une structure Python telle qu'un dictionnaire ou une liste, en fonction de l'organisation des données dans le fichier, facilitant ainsi l'accès aux informations qu'il contient.

Exemple :

Si vous souhaitez charger le fichier ***data.json*** de notre exemple pour utiliser les données qu'il contient, vous pouvez procéder comme suit :

```python
import json

# Ouverture du fichier data.json en mode lecture
with open('data.json', 'r', encoding='utf-8') as f:
    donnees = json.load(f)  # Chargement des données JSON dans une structure Python

# Affichage de toute la structure 'donnees' pour visualiser le contenu chargé
print(donnees)

# Affichage des données chargées
for personne in donnees:
    print("Nom : {}".format(personne['nom']))
    print("Âge : {}".format(personne['age']))
    print("Ville : {}".format(personne['ville']))
    print("Intérêts : ")
    for interet in personne['interets']:
        print("{}".format(interet))
    print()  # Ajoute une ligne vide entre les personnes
```

Sortie :

```
[{'nom': 'Pierre', 'age': 30, 'ville': 'Paris', 'interets':
['lecture', 'sport', 'musique']}, {'nom': 'Marie', 'age': 25,
'ville': 'Lyon', 'interets': ['cuisine', 'voyages',
'photographie']}]
Nom : Pierre
Âge : 30
Ville : Paris
Intérêts :
lecture
sport
musique

Nom : Marie
Âge : 25
Ville : Lyon
Intérêts :
cuisine
voyages
photographie
```

La sortie commence par afficher le contenu complet du fichier ***data.json***, qui est une liste composée de deux éléments (dictionnaires). Chaque élément représente une personne, avec ces informations (son nom, son âge, sa ville et ses centres d'intérêt).
Le code affiche ensuite les détails de chaque personne de manière structurée :

- Le nom, l'âge et la ville sont présentés directement.

- Les centres d'intérêt sont énumérés sous le titre "Intérêts".

Pour plus de détails, consultez la documentation suivante [1] : JSON Load in Python - GeeksforGeeks.

1.3. Autres fonctions natives autorisées

a) **La fonction abs()** est une fonction native de Python, faisant partie de la bibliothèque standard. Elle renvoie la valeur absolue d'un nombre.

Exemple :

```
nombre = -10
print(abs(nombre)) # Affiche 10
```

b) **La fonction lower()** en Python permet de convertir tous les caractères alphabétiques d'une chaîne de caractères en minuscules. Elle fait partie de la bibliothèque standard et est utilisée sur une chaîne pour renvoyer une nouvelle chaîne où tous les caractères sont en minuscules. Cette méthode ne modifie pas la chaîne d'origine, mais renvoie une nouvelle chaîne modifiée.

Exemple :

```
texte = "Bonjour Le Monde"
texte_minuscule = texte.lower()
print(texte_minuscule)
```

Sortie :

```
bonjour le monde
```

c) **La fonction split()** en Python est utilisée pour diviser une chaîne de caractères en plusieurs sous-chaînes en fonction d'un délimiteur spécifié. Par défaut, elle utilise l'espace comme délimiteur, mais on peut aussi spécifier un autre caractère comme séparateur. Elle renvoie une liste contenant les sous-chaînes obtenues.

Exemple1 :

```
texte = "Python est un langage puissant"
mots = texte.split()  # Séparation par défaut, c'est-à-dire par les espaces
print(mots)
```

Sortie :

```
['Python', 'est', 'un', 'langage', 'puissant']
```

Si vous souhaitez séparer la chaîne par un autre caractère, vous pouvez le spécifier comme argument :

Exemple2 :

```
texte = "1 ;2 ;3 ;4 ;5"
elements = texte.split(" ;")  # Séparation par le point-virgule
print(elements)
```

[1] https://www.geeksforgeeks.org/json-load-in-python/

Sortie :

```
['1', '2', '3', '4', '5']
```

2. Création du Projet

2.1. Premiers Pas pour Démarrer et Gérer Votre Projet en Binôme

2.1.1. Instructions sur l'organisation du groupe et l'usage de Git

Ce projet doit être réalisé en **binôme** (un seul trinôme est autorisé uniquement pour les groupes impairs). Une collaboration efficace avec votre binôme est essentielle pour mener à bien ce projet.

Vous utiliserez **Git** et **GitHub** pour gérer le projet. Ces outils vous permettront de partager le code, de versionner les modifications et de suivre l'avancement du projet de manière collaborative. **Avant de commencer, il est indispensable que chaque membre crée un compte sur GitHub.**

Une fois votre compte créé, vous pourrez **consulter le document « Guide d'utilisation de Git dans PyCharm » disponible sur Moodle**. Ce guide a été spécialement préparé pour vous accompagner dans l'utilisation de ces outils.

Assurez-vous de partager les tâches de manière équitable, de communiquer régulièrement et de suivre un planning pour garantir la réussite du projet.

2.1.2. Gestion du dépôt sur GitHub

Pour bien démarrer votre projet, suivez attentivement les étapes ci-dessous :

1. L'un des membres du binôme doit initier le projet en créant le dépôt principal sur son compte **GitHub**. Cette création peut se faire directement sur la plateforme **GitHub** ou via **PyCharm** (référez-vous au guide d'utilisation pour les détails).
2. Le dépôt doit être nommé selon la convention suivante : «**pyfort-nomEtudiant1-nomEtudiant2-nomGroupeTP**».
   Exemple : Pour un binôme composé de **Durant** et **Dupont**, appartenant au groupe **C**, le dépôt doit s'appeler : **pyfort-durant-dupont-c**.
3. Une fois le dépôt principal créé, l'autre membre du binôme (ou trinôme) devra **cloner** ce dépôt sur son ordinateur via **PyCharm**.
4. Celui qui a créé le dépôt principal doit **ajouter l'autre membre du binôme (ou trinôme)** ainsi que **les deux enseignants de TP** comme **collaborateurs** sur le **dépôt GitHub**. Pour ce faire, allez dans les **paramètres du dépôt sur GitHub**, puis dans la section « **Collaborateurs** », et ajoutez les emails ou noms des utilisateurs **GitHub** des membres et enseignants.
5. Tout au long du projet, veillez à effectuer régulièrement des **pulls** pour récupérer les dernières modifications du binôme et à pousser vos propres changements en utilisant

Git **push**. Cela garantit que chaque membre travaille avec la version la plus à jour du code.

2.2. Structure du Projet

Débutez la création de votre projet en structurant correctement vos fichiers et dossiers, conformément à l'arborescence indiquée ci-dessous :

```
.
├── data
│   ├── enigmesPF.json
│   └── indicesSalle.json
├── enigme_pere_fouras.py
├── epreuve_finale.py
├── epreuves_hasard.py
├── epreuves_logiques.py
├── epreuves_mathematiques.py
├── fonctions_utiles.py
└── main.py
```

Le fichier **main.py** servira de point d'entrée principal pour le jeu, en appelant des fonctions spécifiques définies dans le module **fonctions_utiles.py** pour gérer les différentes étapes du jeu. Les fonctions liées aux épreuves seront réparties dans des modules spécifiques comme suit :

- **epreuves_mathematiques.py** : Ce module contiendra les fonctions pour gérer les défis mathématiques, tels que des calculs et la résolution d'équations.
- **epreuves_hasard.py** : Il regroupera les fonctions pour simuler des jeux de hasard, tels que le Bonneteau ou le lancer de dés.
- **epreuves_logiques.py** : Ce module implémentera l'un des jeux logiques, tels que la bataille navale ou le morpion.
- **enigme_pere_fouras.py** : Il contiendra les fonctions pour gérer l'épreuve des énigmes du Père Fouras.
- **epreuve_finale.py** : Ce module contiendra les fonctions permettant de simuler la dernière étape du jeu, où les joueurs doivent deviner un mot-code pour accéder à la salle du trésor.

Les deux dernières épreuves (Père Fouras et salle du trésor) nécessitent les fichiers ***enigmesPF.json*** et ***indicesSalle.json***, disponibles sur Moodle. Vous devez les ***télécharger*** et les ***placer*** dans le répertoire ***data***, car ces fichiers contiennent les données nécessaires pour ces défis.

3. Implémentation

Dans cette section, nous détaillerons l'organisation des fonctions spécifiques à implémenter dans le projet. Voici les points clés à considérer :

- **Structure des fonctions** : Un ensemble de fonctions spécifiques a été défini pour ce projet. Il est important de respecter cette structure pour garantir la cohérence du code, mais vous êtes encouragés à ajouter des fonctions complémentaires selon votre propre logique et pour améliorer la lisibilité et l'efficacité de votre programme.
- **Choix des jeux à implémenter** : Vous aurez à choisir parmi plusieurs jeux selon le type de l'épreuve à implémenter. Chaque jeu est accompagné d'une indication de son niveau de difficulté (faible, moyenne, élevée). Ce niveau est à titre indicatif, car il peut varier en fonction de votre propre logique de programmation et de la manière dont vous abordez chaque tâche. Il est donc conseillé de bien évaluer chaque option avant de faire votre choix.
- **Répartition des tâches** : Une répartition claire des tâches est primordiale. Assurez-vous que chaque membre du groupe comprend bien ses responsabilités et les parties du code qu'il doit implémenter.

Pour faciliter le développement du projet, nous vous conseillons de suivre l'ordre suivant pour l'implémentation des fonctions :

1. **Commencez par les épreuves les plus simples**, à savoir les **épreuves mathématiques** et les **épreuves de hasard**. Ces épreuves sont relativement plus simples à implémenter et vous permettront de tester rapidement les fonctions associées.
2. Poursuivez avec l'implémentation du **jeu logique** que vous aurez choisi, des **énigmes du Père Fouras** et de **l'épreuve finale**. Ces épreuves ajouteront progressivement de la complexité au projet, et vous êtes libre de les aborder dans l'ordre qui vous convient.
3. Finalisez l'implémentation des fonctions dans le module **fonctions_utiles.py**. Certaines de ces fonctions peuvent être développées plus tôt, notamment celles qui facilitent la gestion du jeu et des équipes.
4. Le fichier **main.py** doit être le dernier à être complété, car il coordonne l'appel des différentes fonctions et gère l'exécution du jeu dans son ensemble.

3.1. Le module epreuves_mathematiques.py

Dans le module **epreuves_mathematiques**, vous devrez :

1. Choisir et implémenter **3 épreuves** parmi les 4 listées ci-dessous. Certaines épreuves nécessitent uniquement l'implémentation de la fonction principale, tandis que d'autres demandent également des sous-fonctions spécifiques.
2. Implémenter la fonction **epreuve_math()** qui sélectionnera aléatoirement l'une des épreuves que vous avez implémentées et exécutera la fonction correspondante.

3.1.1. Épreuve de la Factorielle (faible)

1. Implémentez une fonction **factorielle(n)** qui calcule la factorielle de n (notée n!). La factorielle de n est définie comme :
   - $0! = 1$
   - $n! = n \times (n-1) \times \cdots \times 1$ pour $n > 0$

   Par exemple, la factorielle de 4 est $4! = 4 \times 3 \times 2 \times 1 = 24$.

   Exemple :

```
factorielle(5)  # Retourne 120
```

2. Implémentez une fonction **epreuve_math_factorielle()** qui génère un nombre aléatoire **n** entre 1 et 10, puis demande au joueur de calculer la factorielle de **n**. La réponse du joueur est comparée à la valeur calculée par la fonction **factorielle(n)**. Si la réponse est correcte, un message indiquant que le joueur gagne une clé est affiché, et la fonction renvoie **True**. Sinon, la fonction renvoie **False**.

   Exemple d'interface utilisateur :

```
Épreuve de Mathématiques: Calculer la factorielle de 5.
Votre réponse: 120
Correct! Vous gagnez une clé.
```

3.1.2. Épreuve d'Équation Linéaire (faible)

1. Implémentez une fonction **resoudre_equation_lineaire()** qui génère deux nombres aléatoires **a** et **b** entre 1 et 10, puis résout l'équation linéaire **ax + b = 0** pour **x**. Cette fonction renvoie les valeurs de **a**, **b** ainsi que la solution correcte de l'équation, c'est-à-dire la valeur de **x = -b/a**.
2. Implémentez une fonction **epreuve_math_equation()** qui appelle la fonction **resoudre_equation_lineaire()** pour obtenir les valeurs générées ainsi que la solution. La fonction **epreuve_math_equation()** demande au joueur de résoudre l'équation et compare la réponse donnée avec la solution correcte. Elle retourne **True** si la réponse est correcte, sinon **False**.

   Exemple d'interface utilisateur :

```
Épreuve de Mathématiques: Résoudre l'équation 4x + 6 = 0.
Quelle est la valeur de x: -1.5
Correct! Vous gagnez une clé.
```

3.1.3. Épreuve des Nombres Premiers (moyenne)

1. Implémentez deux fonctions :
   - **est_premier(n):** qui vérifie si **n** est un nombre premier. Un nombre est premier s'il est supérieur à 1 et n'a pas de diviseur autre que 1 et lui-même.

- **premier_plus_proche(n):** qui retourne le premier nombre premier supérieur ou égal à **n**.

2. Implémentez la fonction **epreuve_math_premier()** qui génère un nombre aléatoire **n** entre 10 et 20, puis demande au joueur de trouver le nombre premier le plus proche de **n**. La solution correcte doit être obtenue en appelant la fonction **premier_plus_proche(n)**. La fonction retourne ***True*** si la réponse donnée par le joueur est correcte, sinon elle retourne ***False***.

   Exemple d'interface utilisateur :

```
Épreuve de Mathématiques: Trouver le nombre premier le plus proche
de 14.
Votre réponse: 17
Correct! Vous gagnez une clé.
```

3.1.4. Épreuve de la Roulette Mathématique (moyenne)

Implémentez la fonction **epreuve_roulette_mathematique()** qui génère cinq nombres aléatoires entre 1 et 20, puis choisit aléatoirement une opération parmi l'***addition***, la ***soustraction*** et la ***multiplication***.

Les opérations doivent être effectuées directement dans la fonction :

- Si l'opération est une addition, additionnez tous les nombres.
- Si l'opération est une soustraction, effectuez la soustraction de tous les nombres dans l'ordre.
- Si l'opération est une multiplication, multipliez tous les nombres ensemble.

Le joueur doit calculer le résultat de l'opération appliquée aux nombres de la 'roulette' et fournir sa réponse. S'il donne la bonne réponse, il gagne une clé. La fonction retourne **True** si la réponse est correcte, sinon elle retourne **False**.

Exemple d'interface utilisateur :

```
Nombres sur la roulette : [3, 5, 7, 8, 2]
Calculez le résultat en combinant ces nombres avec une addition
Votre réponse : 25
Bonne réponse! Vous avez gagné une clé.
```

3.1.5. Fonction epreuve_math() pour la sélection aléatoire d'une épreuve

Implémentez une fonction **epreuve_math()** qui sélectionne aléatoirement une épreuve parmi plusieurs épreuves mathématiques :

1. Créez une liste **epreuves** et ajoutez-y les fonctions d'épreuves que vous avez implémentées sous forme de références, sans parenthèses, pour pouvoir les appeler

ultérieurement. En Python, cela permet de stocker les fonctions elles-mêmes dans la liste, afin de les exécuter dynamiquement.

2. Utilisez la fonction appropriée de **random** pour sélectionner aléatoirement une fonction dans la liste et la stocker dans la variable **epreuve**.
3. Une fois l'épreuve choisie, appelez cette fonction en exécutant **epreuve()**. Cela permettra d'exécuter dynamiquement l'épreuve sélectionnée et de retourner le résultat de l'épreuve (**True** ou **False**) en fonction de la réponse du joueur.

Cette méthode assure que chaque appel de la fonction **epreuve_math()** exécute une épreuve aléatoire parmi celles disponibles.

3.2. Le module epreuves_hasard.py

Les épreuves de hasard et les épreuves logiques seront contre le maître du jeu. Dans Fort Boyard, ce dernier supervise les défis et guide les joueurs à travers diverses épreuves, tout en étant aussi leur adversaire lors de certaines épreuves. Vous implémenterez des jeux où le joueur s'affronte directement avec le maître du jeu.

Dans le module ***epreuves_hasard,*** vous devez implémenter **les deux épreuves** ainsi que la fonction **epreuve_hasard()**.

3.2.1. Bonneteau (faible)

Le joueur doit deviner sous quel bonneteau (A, B ou C) se cache la clé. Il dispose de deux essais pour le trouver. À chaque essai, la clé est placée aléatoirement sous l'un des bonneteaux. Implémentez la fonction **bonneteau()** en vous basant sur la version approximative de la fonction en pseudo-algorithme suivante :

Fonction : bonneteau() → booléen
Variables locales[2] :

- une liste contenant trois éléments 'A', 'B' et 'C'
- un entier représentant le numéro de la tentative
- une variable stockant la valeur du bonneteau sélectionné aléatoirement
- une variable stockant le choix du joueur

Début

1. Initialiser la liste des bonneteaux
2. Afficher un message de bienvenue en expliquant les règles du jeu et en précisant le nombre d'essais autorisés
3. Afficher les bonneteaux disponibles afin de permettre au joueur de faire son choix
4. **Pour chaque tentative** de 1 à 2, faire :
    5. Choisir aléatoirement la lettre correspondant au bonneteau parmi la liste des bonneteaux et la stocker dans une variable.
    6. Afficher le nombre de tentatives restantes pour le joueur.

[2] **La liste des variables locales est incomplète.**

7. Demander au joueur de choisir un bonneteau (A, B ou C) et convertir le choix en majuscules.
    a. Si le choix est valide (présent dans les bonneteaux), alors:
        - Si le choix correspond à celui choisi aléatoirement :
            - Afficher un message indiquant que la clé a été retrouvé sous le bonneteau.
            - Retourner **Vrai**.
        - Sinon, afficher que le joueur n'a pas réussi cette tentative.
    b. Sinon, si le choix est invalide :
        - L'indiquer avec un message.
8. Incrémenter le nombre de tentatives.

9. **Après les deux essais** :
    - Afficher que le joueur a perdu et révéler sous quel bonneteau la clé se trouvait.
    - Retourner **Faux**.

Fin.

3.2.2. Lancer de dés (moyenne)

Le joueur et le maître du jeu lancent chacun deux dés. Le premier à obtenir un 6 remporte la partie, avec un maximum de trois essais. Implémentez la fonction **jeu_lance_des()** en vous appuyant sur la version approximative en pseudo-algorithme suivante :

Fonction: jeu_lance_des() → booléen

Variables locales[3] **:**

- Une variable (ou constance) représentant le nombre d'essais autorisés (3)
- Un tuple contenant deux valeurs entières, générées aléatoirement entre 1 et 6, représentant les dés du joueur.
- Un tuple contenant deux valeurs entières, générées aléatoirement entre 1 et 6, représentant les dés du maître du jeu.

Début

1. Pour chaque essai de 1 à 3, faire :
    2. Afficher le nombre d'essais restants
    3. Afficher un message invitant le joueur à lancer les dés en appuyant sur la touche "Entrée".
    (*après que le joueur appuis sur la touche "Entrée" la fonction génère les deux nombres aléatoires*)
    4. Stocker dans un tuple deux valeurs générées aléatoirement par la fonction appropriée de la bibliothèque ***random***. Chaque

[3] **La liste des variables locales est incomplète.**

valeur doit être un entier compris entre 1 et 6, représentant les résultats du lancer des dés.

5. Afficher les valeurs obtenues par le joueur.
6. Si l'une des deux valeurs est égale à 6, alors :
   - Afficher que le joueur a remporté la partie et la clé.
   - Retourner **Vrai**

(*après c'est au tour du maître du jeu*)

7. Stocker dans un tuple deux valeurs générées aléatoirement.
8. Afficher les valeurs obtenues par le maître du jeu.
9. Si l'une des deux valeurs est égale à 6, alors :
   - Afficher que le maître du jeu a remporté la partie.
   - Retourner **Faux**
10. Si aucun 6 n'est obtenu :
    - Afficher un message indiquant qu'on passe au prochain essai.

(*Après trois essais, aucun des joueurs n'a gagné*)

11. Afficher qu'aucun joueur n'a obtenu un 6 après trois essais et que c'est un match nul.
12. Retourner **Faux**

Fin

3.2.3. Fonction epreuve_hasard() pour la sélection aléatoire d'une épreuve

Implémentez la fonction **epreuve_hasard()** en suivant le même principe que pour la fonction ***epreuve_math()*** :

1. Créez une liste **epreuves** et ajoutez-y les références des fonctions.
2. Utilisez une fonction de **random** pour sélectionner aléatoirement une fonction parmi celles présentes dans la liste et assignez-la à la variable **epreuve**.
3. Appelez cette fonction sélectionnée en exécutant **epreuve()**.

3.3. Le module epreuves_logiques.py

Vous devez implémenter **l'un des trois jeux proposés**. La note pour cette partie sera attribuée en fonction du jeu choisi. Pour chaque jeu, vous disposerez d'une décomposition des fonctions et d'un exemple d'exécution attendu pour vous guider.

3.3.1. Jeu de NIM ou les bâtonnets (Faible)

Le joueur et le maître du jeu (IA) retirent à tour de rôle 1, 2 ou 3 bâtonnets d'un total de 20. Celui qui retire le dernier bâtonnet perd la partie. Le maître du jeu suit une stratégie basée sur les multiples de 4 pour maximiser ses chances de victoire.

Liste des fonctions à implémenter :

- **affiche_batonnets(n):** prend en paramètre un entier **n**, qui représente le nombre de bâtonnets restants. Elle affiche ce nombre sous forme de barres (|), chaque barre correspondant à un bâtonnet.
- **joueur_retrait(n):** prend en paramètre un entier **n** représentant le nombre de bâtonnets restants. Cette fonction demande au joueur de choisir combien de bâtonnets il souhaite retirer (1, 2 ou 3), en s'assurant que l'entrée est correcte. Elle retourne le nombre de bâtonnets retirés par le joueur.
- **maitre_retrait(n):** Elle prend en paramètre un entier **n**, représentant le nombre de bâtonnets restants. Elle applique une stratégie d'IA basée sur le reste de la division de n par 4. L'IA (le maître du jeu) choisit le nombre de bâtonnets à retirer en fonction de cette stratégie, dans le but de forcer l'adversaire à perdre. Elle retourne le nombre de bâtonnets retirés par le maître du jeu.
- **jeu_nim() :** C'est la fonction principale du jeu des bâtonnets. Elle appellera les fonctions définies précédemment pour gérer le déroulement du jeu de la manière suivante :
    1. Initialise le nombre de bâtonnets à 20.
    2. Utilise une variable booléenne pour indiquer que c'est le tour du joueur de commencer.
    3. Tant que le nombre de bâtonnets est différent de zéro :
    4. Affiche les bâtonnets restants.
    5. Une itération représente un tour, soit celui du joueur, soit celui du maître du jeu.
    6. Le nombre de bâtonnets retirés est récupéré par l'appel de la fonction correspondante.
    7. Le nombre de bâtonnets restants est mis à jour et affiché à chaque tour.
    8. Le jeu continue (avec alternance des joueurs) jusqu'à ce qu'il ne reste plus de bâtonnets.
    9. Celui qui retire le dernier bâtonnet est déclaré perdant.
    10. La fonction devra retourner Vrai si le joueur gagne, faux sinon.

Exemple du résultat de l'exécution du jeu :

```
Bâtonnets restants: ||||||||||||||||||||
Tour du joueur.
Combien de bâtonnets voulez-vous retirer (1, 2 ou 3) ? 3
Bâtonnets restants: ||||||||||||||||
Le maître du jeu retire 1 bâtonnets.
Bâtonnets restants: ||||||||||||||||
Tour du joueur.
Combien de bâtonnets voulez-vous retirer (1, 2 ou 3) ? 3
Bâtonnets restants: ||||||||||||
Le maître du jeu retire 1 bâtonnets.
Bâtonnets restants: ||||||||||||
Tour du joueur.
Combien de bâtonnets voulez-vous retirer (1, 2 ou 3) ? 3
```

```
Bâtonnets restants: |||||||||
Le maître du jeu retire 1 bâtonnets.
Bâtonnets restants: ||||||||
Tour du joueur.
Combien de bâtonnets voulez-vous retirer (1, 2 ou 3) ? 3
Bâtonnets restants: |||||
Le maître du jeu retire 1 bâtonnets.
Bâtonnets restants: ||||
Tour du joueur.
Combien de bâtonnets voulez-vous retirer (1, 2 ou 3) ? 3
Bâtonnets restants: |
Le maître du jeu retire 1 bâtonnets.
Le maître du jeu a retiré le dernier bâtonnet. Le joueur gagne !
```

3.3.2. Le Morpion ou tic-tac-toe (Moyenne)

Le joueur et le maître du jeu (IA) s'affrontent dans une partie classique de morpion. Le premier à aligner trois symboles identiques (*horizontalement*, *verticalement*, ou en *diagonale*) gagne la partie. Le maître du jeu utilise une stratégie simple pour tenter de gagner ou bloquer le joueur.

Liste des fonctions à implémenter :

- **afficher_grille(grille)** : La fonction prend en paramètre une grille 3x3, représentée par une liste 2D, et l'affiche. Chaque case de la grille peut être vide (" ") ou contenir un symbole ('X' ou 'O'), avec un séparateur entre les lignes.
- **verifier_victoire(grille, symbole) :** La fonction prend en paramètre une liste 2D grille, et une chaîne de caractères symbole ('X' ou 'O') à vérifier. Elle examine les lignes, colonnes et diagonales de la grille pour savoir si le symbole a gagné, retournant *True* en cas de victoire, sinon *False*.
- **coup_maitre(grille, symbole) :** La fonction prend en paramètre une liste 2D grille, représentant l'état actuel de la grille de jeu, et une chaîne de caractères symbole, représentant le symbole du maître du jeu. Cette fonction détermine le coup du maître du jeu en priorité pour gagner, puis pour bloquer le joueur si nécessaire, et enfin joue un coup au hasard si aucune de ces actions n'est requise. Elle retourne un tuple (ligne, colonne) correspondant aux coordonnées de la case choisie.
- **tour_joueur(grille) :** La fonction prend en paramètre une liste 2D grille, représentant l'état actuel de la grille de jeu. Cette fonction ne retourne rien, mais elle modifie directement la grille en permettant au joueur (symbole 'X') de choisir une case vide pour y déposer son symbole. Le joueur doit saisir les coordonnées de la case sous la forme ligne, colonne. Avant de placer le symbole, la fonction vérifie que la case choisie est bien vide. Si la case est déjà occupée, le joueur devra choisir à nouveau.
- **tour_maitre(grille) :** La fonction prend en paramètre une liste 2D grille, représentant l'état actuel de la grille de jeu. Cette fonction ne retourne rien, mais elle modifie directement la grille en plaçant le coup du maître du jeu ('O'). Le maître du jeu joue en suivant la stratégie définie dans la fonction ***coup_maitre()***.

- **grille_complete(grille) :** La fonction prend en paramètre une liste 2D grille, représentant l'état actuel de la grille de jeu. Elle retourne *True* si la grille est complète (aucune case vide), sinon *False*. Cette fonction sera utilisée pour vérifier si le jeu est terminé par un match nul ou s'il peut continuer.
- **verifier_resultat(grille) :** La fonction prend en paramètre une liste 2D grille, représentant l'état actuel de la grille de jeu. Elle retourne *True* si la partie est terminée, c'est-à-dire si le joueur 'X' ou le maître du jeu (joueur 'O') a gagné, ou si la partie est terminée par un match nul, en vérifiant les résultats avec les fonctions **verifier_victoire()** et **grille_complete()**. Si aucune condition de fin de jeu n'est remplie, elle retourne *False*, ce qui signifie que la partie continue.
- **jeu_tictactoe() :** C'est la fonction principale qui orchestre l'ensemble du jeu. Elle ne prend aucun paramètre et retourne *True* si le joueur 'X' remporte la partie, *False* sinon (c'est-à-dire si c'est le maître du jeu ou un match nul). Voici les étapes :
  1. La grille (Liste 2D) est initialisée avec des espaces vides.
  2. Une boucle est définie permettant d'alterner les tours entre le joueur et le maître du jeu :
     1) La fonction **tour_joueur()** est appelée pour gérer le coup du joueur. Elle permet au joueur de placer son symbole ('X') sur la grille.
     2) La fonction **verifier_resultat()** est appelée pour vérifier si la partie est terminée (victoire ou match nul). Si le joueur 'X' gagne, la fonction retourne **True**.
     3) Si la partie n'est pas terminée après le tour du joueur, la fonction **tour_maitre()** est appelée pour que le maître du jeu (symbole 'O') fasse son coup.
     4) La fonction **verifier_resultat()** est appelée à nouveau pour vérifier si un joueur a gagné ou si la grille est complète. Si le maître du jeu gagne ou si un match nul est détecté, la fonction retourne **False**.
  3. La boucle continue jusqu'à ce qu'un joueur gagne ou que la grille soit complète, indiquant un match nul. Si l'un de ces événements se produit, le jeu se termine.

Exemple du résultat de l'exécution du jeu :

```
Pour jouer, entrez les coordonnées de votre coup sous forme ligne, colonne
(ex: 1,2)
  |   |
---------
  |   |
---------
  |   |
---------
Joueur X, c'est à vous. Où voulez-vous placer votre symbole ? 1,1
X |   |
---------
  |   |
```

```
---------
  |   |
---------
Tour de du maître du jeu (O)...
X |   |
---------
  | O |
---------
  |   |
---------
Joueur X, c'est à vous. Où voulez-vous placer votre symbole ? 3,3
X |   |
---------
  | O |
---------
  |   | X
---------
Tour de du maître du jeu (O)...
X |   |
---------
  | O |
---------
O |   | X
---------
Joueur X, c'est à vous. Où voulez-vous placer votre symbole ? 1,3
X |   | X
---------
  | O |
---------
O |   | X
---------
Tour de du maître du jeu (O)...
X | O | X
---------
  | O |
---------
O |   | X
---------
Joueur X, c'est à vous. Où voulez-vous placer votre symbole ? 2,3
X | O | X
---------
  | O | X
---------
O |   | X
---------
Le joueur a gagné !
```

3.3.3. Jeu de bataille navale (Élevée)

Le joueur place deux bateaux sur une grille et tente de deviner la position des deux bateaux de son adversaire (le maître du jeu). Le gagnant est celui qui réussit à toucher tous les bateaux adverses. Chaque joueur dispose de deux grilles : une pour la position de ses bateaux et une autre pour enregistrer ses tirs. Le jeu alterne les tours entre les

joueurs, chacun cherchant à toucher les bateaux de l'autre. Le jeu se termine lorsque les deux bateaux d'un joueur sont coulés.

Liste des fonctions à implémenter :

- **suiv(joueur)** : La fonction prend en paramètre l'indice d'un joueur actuel (0 ou 1) et renvoie l'indice du joueur suivant, soit 0 ou 1.
- **grille_vide() :** La fonction génère et retourne une grille vide sous forme de liste 2D de taille 3x3. Chaque case de la grille est initialisée avec un espace vide (" ").
- **affiche_grille(grille, message) :** La fonction prend en paramètres une grille (liste 2D) et une chaîne de caractères représentant un message. Ce message peut indiquer, par exemple, « *Rappel de l'historique des tirs que vous avez effectués :* » ou « *Découvrez votre grille de jeu avec vos bateaux :* ». Cette fonction est utilisée pour visualiser soit l'historique des tirs du joueur, soit les positions des bateaux qu'il a placés en début de partie. Elle affiche d'abord le message, suivi de la grille, ligne par ligne, avec des séparateurs (|) entre les cases et une bordure sous la grille.
- **demande_position():** La fonction demande à l'utilisateur de saisir une position valide sur la grille. Vérifie les limites de la position entrée, puis retourne un tuple (ligne, colonne) correspondant.
- **init()** : La fonction crée une grille vide pour les bateaux d'un joueur. Elle demande à l'utilisateur de saisir la position des bateaux sous la forme ligne, colonne. Si la position est valide et libre, un bateau est placé à cette case en y mettant la lettre 'B'. L'utilisateur doit placer deux bateaux. La fonction retourne la grille du joueur avec les deux bateaux placés.
- **tour(joueur, grille_tirs_joueur, grille_adversaire) :** Cette fonction gère un tour de jeu, où un joueur **ou** le maître du jeu effectue un tir. Elle prend en paramètres :
  - joueur : Indice du joueur (0 ou 1).
  - grille_tirs_joueur : Grille des tirs du joueur.
  - grille_adversaire : Grille des bateaux de l'adversaire.

  Elle affiche l'historique des tirs, demande une position au joueur (ou génère une position aléatoire pour le maître du jeu), et met à jour la grille des tirs avec 'x' pour un coup réussi et '.' pour un coup manqué. La fonction ne retourne rien mais modifie les grilles du joueur et de l'adversaire.

- **gagne(grille_tirs_joueur)** : La fonction prend en paramètre la grille des tirs du joueur. Elle vérifie si tous les bateaux de l'adversaire ont été coulés en comptant le nombre de cases marquées par 'x' sur la grille des tirs. La fonction retourne *True* si tous les bateaux ont été coulés, sinon elle retourne *False*.
- **jeu_bataille_navale() :** Cette fonction gère le déroulement complet du jeu de bataille navale. Elle permet d'effectuer les étapes suivantes :
  1. Afficher un message qui explique les règles du jeu, précisant que chaque joueur doit placer 2 bateaux sur une grille 3x3.

2. La grille des bateaux du joueur est initialisée via la fonction **init()**, et ses bateaux sont affichés.
3. La grille des bateaux du maître du jeu est vide, et deux bateaux sont placés aléatoirement sur cette grille.
4. Les grilles de tirs des deux joueurs (joueur et maître du jeu) sont initialisées et vides.
5. Une boucle est définie permettant d'alterner les tours entre le joueur humain et le maître du jeu :
    a. Le jeu commence avec le joueur (joueur 0).
    b. Le joueur et le maître du jeu alternent les tours. À chaque tour, la fonction **tour()** est appelée pour effectuer un tir.
    c. Après chaque tour, la fonction **gagne()** vérifie si le joueur ou le maître du jeu a coulé tous les bateaux adverses.
    d. Si un joueur a gagné, le message de victoire est affiché et la fonction retourne **True** ou **False** (si le maître du jeu gagne).
    e. Si aucun joueur n'a gagné, le jeu continue avec le changement de tour, en alternant entre le joueur et le maître du jeu grâce à la fonction **suiv()**.
6. Le jeu se termine dès qu'un joueur a coulé tous les bateaux adverses.

Exemple du résultat de l'exécution du jeu :

```
Chaque joueur doit placer 2 bateaux sur une grille de 3x3.
Les bateaux sont représentés par 'B' et les tirs manqués par '.'. Les bateaux
coulés sont marqués par 'x'.
Positionnez vos bateaux :
Bateau 1
Entrez la position (ligne,colonne) entre 1 et 3 (ex: 1,2) : 1,2
Bateau 2
Entrez la position (ligne,colonne) entre 1 et 3 (ex: 1,2) : 1,1
Découvrez votre grille de jeu avec vos bateaux :
| B | B |   |
|   |   |   |
|   |   |   |
-------------

C'est à votre tour de faire feu !:
Rappel de l'historique des tirs que vous avez effectués :
|   |   |   |
|   |   |   |
|   |   |   |
-------------
Entrez la position (ligne,colonne) entre 1 et 3 (ex: 1,2) : 1,2
Touché coulé !

C'est le tour du maître du jeu :
Le maître du jeu tire en position 1,1
Touché coulé !

C'est à votre tour de faire feu !:
```

```
Rappel de l'historique des tirs que vous avez effectués :
|   | x |   |
|   |   |   |
|   |   |   |
-------------
Entrez la position (ligne,colonne) entre 1 et 3 (ex: 1,2) : 1,1
Dans l'eau...

C'est le tour du maître du jeu :
Le maître du jeu tire en position 1,3
Dans l'eau...

C'est à votre tour de faire feu !:
Rappel de l'historique des tirs que vous avez effectués :
| . | x |   |
|   |   |   |
|   |   |   |
-------------
Entrez la position (ligne,colonne) entre 1 et 3 (ex: 1,2) : 1,3
Dans l'eau...

C'est le tour du maître du jeu :
Le maître du jeu tire en position 2,2
Dans l'eau...

C'est à votre tour de faire feu !:
Rappel de l'historique des tirs que vous avez effectués :
| . | x | . |
|   |   |   |
|   |   |   |
-------------
Entrez la position (ligne,colonne) entre 1 et 3 (ex: 1,2) : 2,2
Dans l'eau...

C'est le tour du maître du jeu :
Le maître du jeu tire en position 3,3
Dans l'eau...

C'est à votre tour de faire feu !:
Rappel de l'historique des tirs que vous avez effectués :
| . | x | . |
|   | . |   |
|   |   |   |
-------------
Entrez la position (ligne,colonne) entre 1 et 3 (ex: 1,2) : 2,1
Touché coulé !
Le joueur a gagné !
```

3.4. Le module enigme_pere_fouras.py

L'épreuve de l'énigme du Père Fouras est une épreuve mythique du jeu télévisé Fort Boyard, où le joueur doit résoudre une énigme posée par le Père Fouras, le gardien du fort.

Cette épreuve teste la capacité de réflexion et de logique du joueur, qui doit répondre correctement à une question pour obtenir un indice ou une clé.

Dans cette optique, ce module vise à reproduire cette épreuve en utilisant les énigmes extraites du fichier '***enigmesPF.json***'. L'objectif est d'implémenter deux fonctions pour simuler une rencontre avec le Père Fouras, où la fonction présente une énigme au joueur. Le joueur dispose de trois tentatives pour deviner la réponse correcte. S'il répond correctement, il gagne et obtient une clé. En cas de trois tentatives incorrectes, un message de défaite sera affiché avec la solution de l'énigme.

Liste des fonctions à implémenter :

- **charger_enigmes(fichier) :** Cette fonction prend en paramètre le chemin du fichier ***'enigmesPF.json'***. Elle ouvre ce fichier en mode lecture, en charge le contenu, puis le convertit en une liste de dictionnaires JSON. Chaque dictionnaire doit représenter une énigme, avec des paires clé-valeur correspondant à la question et à sa réponse. La fonction retourne ensuite cette liste de dictionnaires.
- **enigme_pere_fouras() :** La fonction simule une rencontre avec le Père Fouras, qui présente une énigme au joueur. Le joueur dispose de trois tentatives pour deviner la réponse correcte. Voici un pseudocode de cette fonction, que vous pouvez utiliser comme référence pour implémenter correctement **enigme_pere_fouras()**.

  Fonction : enigme_pere_fouras() → booléen

  Variables locales[4] **:**
  - Une liste pour stocker les énigmes.
  - Une variable de type dictionnaire pour stocker l'énigme choisie aléatoirement.
  - Une variable de type entier initialisée à 3 pour représenter le nombre d'essais.

  Début :
  1. Utiliser la fonction ***charger_enigmes(fichier)*** pour récupérer les énigmes et les stocker dans la variable de type liste.
  2. Choisir une énigme aléatoirement parmi la liste et la stocker dans la variable correspondante.
  3. Afficher la question de l'énigme choisie.
  4. Tant que le nombre d'essai est supérieur à zéro 0 faire :
     - Demander au joueur de saisir une réponse et la convertir en minuscules.
     - Si la réponse est égale à la réponse attendue de l'énigme:
       - Afficher que la réponse est correcte et que le joueur gagne la clé.
       - Retourner **Vrai.**

[4] **La liste des variables locales est incomplète.**

- Sinon :
- Réduire le nombre d'essais restants de 1.
    - Si des essais restent :
        - Afficher que la réponse est incorrecte, ainsi que le nombre d'essais restants.
    - Sinon :
        - Afficher que le joueur a échoué à l'énigme et la solution.
        - Retourner **Faux.**

Fin.

3.5. Le module epreuve_finale.py

La salle du trésor représente la dernière étape du jeu, où l'équipe, après avoir collecté toutes les clés, doit déchiffrer le code nécessaire pour ouvrir la porte et accéder au trésor. Dans ce module, vous devez simuler cette étape finale pour accéder à la salle du trésor de Fort Boyard.

Implémentez la fonction **salle_De_Tresor()** en suivant les étapes ci-dessous :

1. Les données sont d'abord chargées depuis le fichier ***'indicesSalles.json'***, qui contient les indices et le mot-code nécessaires. Il est important de vérifier la structure de ce fichier pour comprendre comment lire et accéder aux données correctement.
2. Un code d'émission est ensuite choisi aléatoirement, et les indices ainsi que le mot-code associé sont extraits.
3. Les trois premiers indices sont affichés pour aider le joueur à deviner le mot-code, et il dispose de trois essais au total.
4. Après chaque mauvaise réponse, le joueur reçoit un nouvel indice et le nombre d'essais restants est indiqué.
5. Si le joueur échoue après trois essais, le mot-code correct est révélé.
6. Un message final indique ensuite si le joueur a réussi ou échoué.

Voici un pseudocode de cette fonction, que vous pouvez utiliser comme guide pour implémenter correctement ***salle_De_Tresor()*.**

```
Fonction : salle_De_Tresor()
Variables locales[5] :
    jeu_tv, emission: Dictionnaire
    indices: Liste
    annee, mot_code: Chaîne de caractères
    essais: Entier
    reponse_correcte: Booléen
```

[5] **La liste des variables locales est incomplète.**

Début

1. Charger les données du fichier ***'indicesSalle.json'*** dans la variable ***jeu_tv***.
2. Obtenir la liste des années disponibles dans ***jeu_tv*** et en sélectionner une aléatoirement, à stocker dans la variable ***annee***.
3. Extraire les émissions associées à cette ***annee*** dans ***jeu_tv***, en sélectionner une aléatoirement, et la stocker dans la variable ***emission***.
4. Extraire les indices et les stocker dans la variable ***indices***, et le mot-code correspondant dans la variable ***mot_code*** pour cette émission.
5. Afficher les trois premiers indices.
6. Initialiser le compteur d'essais à 3.
7. Initialiser une variable ***reponse_correcte*** à faux.
8. Tant que le nombre d'essais est supérieur à 0 :
    - Demander au joueur d'entrer une réponse
    - Si la réponse du joueur est égale au mot-code :
        - Mettre à jour la valeur de ***reponse_correcte***
        - Sortir de la boucle
    - Sinon
        - Décrémenter le compteur d'essais
        - Si le nombre d'essais est supérieur à 0 :
            - Afficher le nombre d'essais restants
            - Afficher un indice supplémentaire
        - Sinon
            - Dévoiler le mot-code correct
9. Si ***reponse_correcte*** est vrai
    - Afficher un message de victoire
10. Sinon
    - Afficher un message d'échec

Fin.

3.6. Le module fonctions_utiles.py

Dans le module **fonctions_utiles.py**, vous devrez implémenter plusieurs fonctions qui géreront les joueurs, les épreuves, et l'enregistrement des résultats du jeu. Ces fonctions seront utilisées pour organiser et suivre le déroulement du jeu.

Les prototypes des fonctions sont fournis ci-dessous. Vous devez implémenter ces fonctions en respectant leur signature et leur comportement, comme décrit.

3.6.1. Fonction introduction()

Implémentez une fonction **introduction()** qui affiche un message de bienvenue pour le jeu et explique les règles de base :

- Le joueur doit accomplir des épreuves pour gagner des clés et déverrouiller la salle du trésor.
- L'objectif est de ramasser trois clés pour accéder à la salle du trésor.

3.6.2. Fonction composer_equipe()

Implémentez la fonction **composer_equipe()** qui permet de créer une équipe de joueurs pour le jeu, sous forme d'une **liste**. L'équipe peut comporter jusqu'à 3 joueurs.

La fonction doit d'abord demander à l'utilisateur combien de joueurs il souhaite inscrire dans l'équipe. Si le nombre de joueurs dépasse 3, un message d'erreur doit être affiché et la saisie doit être redemandée.

Pour chaque joueur, la fonction doit demander et saisir les informations suivantes : son ***nom***, sa ***profession***, et si ce joueur est le ***leader*** de l'équipe. Chaque joueur sera représenté sous la forme d'un **dictionnaire** contenant ces trois informations ainsi qu'un quatrième champ '*cles_gagnees*' initialisé à zéro.

Si, à la fin de la composition de l'équipe, aucun joueur n'a été désigné comme leader, le ***premier joueur*** de l'équipe deviendra automatiquement le ***leader***.

La fonction doit retourner la liste contenant les joueurs qui composent l'équipe.

3.6.3. Fonction menu_epreuves()

Implémentez la fonction **menu_epreuves()** qui affiche le menu suivant permettant à l'utilisateur de choisir parmi différents types d'épreuves disponibles :

```
1. Épreuve de Mathématiques
2. Épreuve de Logique
3. Épreuve du hasard
4. Énigme du Père Fouras
Choix:
```

L'utilisateur saisit le numéro correspondant à l'épreuve, et la fonction retourne ce choix.

3.6.4. Fonction choisir_joueur(equipe)

Implémentez la fonction choisir_joueur(equipe) qui prend en paramètre une liste de dictionnaires représentant les joueurs et permet à l'utilisateur de sélectionner un joueur de l'équipe pour participer à une épreuve.

La fonction doit afficher la liste des joueurs avec leur nom, profession et rôle (afficher "Membre" si ce n'est pas le leader). ***Par exemple :***

```
1. Jean Dupont (Ingénieur) - Leader
2. Marie Martin (Enseignante) - Membre
3. Paul Durand (Docteur) – Membre
Entrez le numéro du joueur:
```

L'utilisateur entre ensuite le numéro du joueur souhaité. La fonction retourne le joueur sélectionné, sous forme de dictionnaire avec ses informations.

3.6.5. Fonction bonus enregistrer_historique(?)

Cette fonction d'enregistrement de l'historique est facultative, mais elle pourra vous rapporter des points bonus. Vous avez la liberté de définir les paramètres à enregistrer selon ce que vous souhaitez sauvegarder dans le fichier ***'output/historique.txt'***, comme le nom de l'épreuve, du joueur, le résultat, ou le nombre de clés obtenues, etc. L'essentiel est de structurer et formater les données de manière claire et lisible.

3.7. Implémentation de la fonction principale dans main.py

Dans le fichier **main.py**, définissez la fonction **jeu()** qui centralisera toutes les actions du jeu en utilisant les fonctions des autres modules que vous avez créés dans ce projet.

Précision : *En lançant cette simulation du jeu, l'utilisateur compose d'abord l'équipe des joueurs, puis prend les décisions tout au long du jeu. Il incarne également le joueur sélectionné, participe à l'épreuve et doit la remporter.*

La fonction **jeu()** inclure les étapes suivantes :

1. **Introduction et composition de l'équipe** : Le jeu commence par une introduction annonçant les règles du jeu, puis l'utilisateur est invité à créer une équipe de joueurs. L'équipe peut être composée de maximum 3 joueurs.
2. **Boucle des épreuves** : Tant que l'équipe n'a pas gagné trois clés, le jeu continue dans une boucle :
    - Un menu des types d'épreuves est affiché et l'utilisateur choisit le type.
    - Une fois le type de l'épreuve validé, la liste des joueurs est affichée et l'utilisateur devra faire son choix.
    - Une épreuve est lancée, choisie de manière aléatoire, à l'exception de l'épreuve de logique.
    - L'utilisateur, incarnant le joueur sélectionné, tente de remporter le jeu.
    - Si l'utilisateur gagne l'épreuve, une clé est obtenue et cela est représenté par l'incrémentation du nombre de clés gagnées pour le joueur sélectionné.
3. **Épreuve finale** : Lorsque les trois clés sont obtenues, l'épreuve finale de la salle du trésor est lancée. L'utilisateur disposera des indices nécessaires pour deviner le mot code et ainsi débloquer la salle. S'il réussit, l'équipe gagne le jeu, sinon elle perd.

Enregistrement des résultats (partie bonus) : Tout au long du jeu, l'historique des épreuves (épreuve sélectionnée, joueur, résultat, etc.) est enregistré. À la fin du jeu, cet historique sera sauvegardé dans le fichier **'output/*historique.txt*'**.

4. Évaluation du Projet : Critères et Attentes

L'évaluation du projet est composée de :

- Une note de collaboration via Git
- Une note sur la documentation
- Une note du code source
- Une note de soutenance.

4.1. Dépôt Git : 10% de la note projet

Preuve de collaboration de tous les membres de l'équipe projet à travers des « commits » étalés sur la période de réalisation du projet.

4.2. Documentation du projet : 10% de la note projet

Cette documentation fait référence au fichier **README.md** du dépôt **Git** du projet. Elle doit suivre la structure suivante :

1. **Présentation Générale :**
    - **Titre du Projet**
    - **Contributeurs :** Liste des membres du binôme avec leurs rôles respectifs.
    - **Description :** Un bref aperçu du projet
    - **Fonctionnalités Principales :** Liste des fonctionnalités clés que l'application offre.
    - **Technologies Utilisées :** Mentionner les langages de programmation, les bibliothèques (si applicable), et les outils.
    - **Installation :**
        - Instructions pour cloner le dépôt Git.
        - Étapes pour configurer l'environnement de développement (installations nécessaires).
    - **Utilisation :**
        - Instructions sur comment exécuter l'application.
        - Éventuellement, des exemples de commandes ou des cas d'utilisation.
2. **Documentation Technique**
    - **Algorithme du jeu :**
        - Présentez l'algorithme que vous avez développé pour le projet en incluant chaque étape sous forme de liste numérotée.
    - **Détails des fonctions implémentées :**
        - Fournissez la liste des prototypes des fonctions, accompagnée d'explications brèves sur leur rôle et de descriptions des paramètres.
    - **Gestion des Entrées et Erreurs :**
        - Décrivez comment le code traite les valeurs et les intervalles, ainsi que les méthodes mises en place pour gérer les erreurs potentielles.
        - Fournissez une liste des bugs connus.
3. **Journal de Bord**
    - Un journal de bord peut aider à suivre le progrès du projet et la répartition des tâches :

- **Chronologie du Projet :** Dates et descriptions des étapes clés, des décisions prises, et des problèmes rencontrés.
- **Répartition des Tâches :** Qui a travaillé sur quoi et comment les tâches ont été divisées.

4. **Tests et Validation**
   - **Stratégies de Test :**
     - Cas de test spécifiques et résultats.
     - Captures d'écran montrant les tests en action.

4.3. Code : 60% de la note projet

L'appréciation du code se fera à deux niveaux :

Sur le fond :

- Le nombre de fonctionnalités réalisées parmi celles qui sont demandées
- Le programme est modulaire :
  - Des fonctions sont utilisées pour répartir les traitements de manière logique et sur différents fichiers afin d'augmenter la lisibilité et la compréhension du code.
- Les algorithmes choisis sont « raisonnablement » efficaces. Ils évitent les traitements et les variables inutiles.
- Le code est robuste par rapport aux erreurs possibles :
  - Il prévoit notamment les saisies sécurisées de l'utilisateur et évite que celles-ci n'entrainent des erreurs d'exécution.
- L'ergonomie de l'interface utilisateur

Sur la forme :

- Chaque fichier de code (**.py**) comporte un commentaire global en tête de fichier, avec le nom du projet, les auteurs et le rôle de ce fichier dans le projet.
- Un commentaire est présent en tête de chaque fonction. Il décrit pour chaque fonction :
  - Son rôle, la signification de ses paramètres, le résultat retourné (s'il n'y en a pas : expliquer pourquoi).
- A l'intérieur des fonctions, des commentaires signalant **uniquement** les **principales** étapes de l'algorithme, et expliquant les parties délicates du code.
- Les fonctions et variables sont nommées de manière cohérente (soit tout en anglais, soit tout en français).

4.4. Soutenance : 20% de la note du projet

La soutenance dure environ **15 minutes** par équipe. Voici ci-dessous quelques éléments d'appréciation de la soutenance :

- Démonstration du travail effectué : Prévoir un plan de tests avec différents scénarios pour couvrir l'ensemble des fonctionnalités réalisées.
- Qualité des réponses aux questions
- Distribution des rôles entre les membres de l'équipe.

4.5. Points Bonus

- Parties signalées comme bonus dans le sujet du projet
- Toute forme d'originalité et d'innovation

5. Remise du travail sur Moodle

Les dépôts du projet doivent être réalisés sur la zone de dépôt dédiée sur Moodle. Un seul membre du binôme effectuera le dépôt.

- **Dépôt intermédiaire : Le samedi 21/12/2024, à midi.** Le fichier ***.zip*** doit inclure tous les éléments suivants :
  - Le **formulaire de dépôt intermédiaire** dûment rempli
  - Tous les fichiers .py déjà créés dans le projet
  - Tous les fichiers .json et/ou .txt utilisés dans le projet
- **Dépôt final : Le dimanche 05/01/2025, à midi.** Il est impératif d'inclure les éléments suivants dans un fichier ***.zip*** :
  - Le **formulaire du dépôt final** dûment rempli
  - Tous les fichiers .py, .json et/ou .txt du projet, structurés selon l'arborescence demandée.
  - Le fichier **README.md** respectant le format demandé

Attention :

- Tout **manquement** de **fichiers** entraînera une note de **0** pour cette partie. Il est donc crucial de vérifier minutieusement que tous les fichiers sont inclus et de ne pas attendre la dernière minute pour effectuer le dépôt.
- **Après la clôture** de la zone de dépôt sur Moodle, **AUCUN envoi** par mail ou par Teams **ne sera accepté**.
- Toute forme de **plagiat** est **formellement interdite**, y compris le copier/coller de sources externes ou la **génération** de code à l'aide **d'intelligences artificielles génératives**. Les membres du groupe peuvent utiliser des sources externes uniquement à titre d'inspiration ou pour stimuler des idées. Cependant, ils doivent veiller à produire un travail personnel et original. Tout **travail** jugé **non original** fera

l'objet d'une **sanction immédiate** sous la forme d'une note de 0/20 et entraînera une **convocation en conseil de discipline**. Pour mieux comprendre la politique anti-plagiat ainsi que l'utilisation d'outils comme Compilatio et Studium, nous vous invitons à consulter la documentation disponible dans l'espace "Direction des études" du portail pédagogique.