

TP — Le jeu du pendu - Correction

```
mot = "parapluie"
mot_cache = "_" * len(mot)
liste_mot_cache = list(mot_cache)
nb_tentatives = 10
deja_proposees = set()

while mot != mot_cache and nb_tentatives > 0:
    print(mot_cache, f"{nb_tentatives} tentatives")
    lettre_saisie = input("Saisissez une lettre: ") # a

    if lettre_saisie in deja_proposees:
        print(f"Vous avez déjà saisi {lettre_saisie}")
        nb_tentatives -= 1
        continue

    deja_proposees.add(lettre_saisie)
    if lettre_saisie not in mot:
        nb_tentatives -= 1
        continue

    for index, lettre in enumerate(mot):
        if lettre == lettre_saisie:
            liste_mot_cache[index] = lettre_saisie

    mot_cache = "".join(liste_mot_cache)

if mot == mot_cache:
    print("Victoire")
else:
    print(f"Perdu, le mot était {mot}")
```

Dans cette correction du jeu du pendu, on adopte une approche dite **"fail fast"**, c'est-à-dire qu'on fait échouer le plus tôt possible chaque cas non valide, sans imbriquer inutilement les conditions. Cela rend le code plus **lisible**, **court** et **facile à maintenir**.

1. L'idée du "fail fast"

Le principe du "fail fast" consiste à **sortir immédiatement d'une situation invalide** au lieu d'ajouter des **else**, des conditions imbriquées ou des cascades de **if**. Ici, chaque situation non conforme (lettre déjà proposée, lettre absente du mot) est traitée **dès qu'elle est détectée**, puis on **continue la boucle** pour passer au tour suivant.

Cela évite des blocs du type :

```
if lettre_saisie in deja_proposees:
    ...
```

```
else:
    if lettre_saisie not in mot:
        ...
    else:
        # le code utile
```

Ce genre de structure devient vite illisible. Grâce au “fail fast”, on garde une **seule couche logique principale** dans la boucle.

2. Les cas d'erreur traités immédiatement

Regardons les deux vérifications critiques :

```
if lettre_saisie in deja_proposees:
    print(f"Vous avez déjà saisi {lettre_saisie}")
    nb_tentatives -= 1
    continue
```

-> Ici, on détecte une erreur de saisie (lettre déjà donnée). On affiche un message, on retire une tentative, puis on saute directement au tour suivant. Aucune autre ligne n'est exécutée pour ce tour.

```
if lettre_saisie not in mot:
    nb_tentatives -= 1
    continue
```

-> Même logique : si la lettre n'est pas dans le mot, inutile de continuer la boucle. On pénalise le joueur et on passe à la suite sans exécuter le reste du code.

Ces deux vérifications permettent d'éviter des **else** imbriqués et de concentrer la “vraie logique” du tour sur le cas **où la lettre est correcte**.

3. Pourquoi transformer le mot caché en liste ?

En Python, une chaîne (**str**) est **immutable** : on ne peut pas remplacer un caractère directement. Par exemple, ceci ne fonctionne pas :

```
mot_cache[2] = "a" # Erreur : les chaînes ne sont pas modifiables
```

Pour pouvoir **révéler** les lettres trouvées au bon endroit, il faut donc **explorer la chaîne en liste** :

```
liste_mot_cache = list(mot_cache)
```

Ainsi, on peut facilement modifier les caractères par index :

```
for index, lettre in enumerate(mot):  
    if lettre == lettre_saisie:  
        liste_mot_cache[index] = lettre_saisie
```

Puis on **reconstruit la chaîne** à la fin du tour :

```
mot_cache = "".join(liste_mot_cache)
```

Cette approche est simple et efficace pour "mettre à jour" un mot affiché dans le terminal.

4. Une boucle principale claire et linéaire

Grâce à ces choix, la boucle **while** reste facile à lire :

1. On affiche l'état du jeu.
2. On récupère la lettre du joueur.
3. On élimine immédiatement les cas d'erreur.
4. On traite ensuite uniquement le cas valide.

On ne se perd jamais dans des **if** imbriqués ou des branchements multiples. Chaque étape est indépendante et clairement délimitée.

5. En résumé

- **Fail fast** : on traite les cas invalides immédiatement et on continue, sans **else**.
- **Lisibilité** : le cœur de la boucle reste concentré sur le cas utile.
- **Simplicité** : on convertit la chaîne en liste pour injecter facilement les lettres trouvées.
- **Maintenance facile** : chaque cas (déjà proposé, faux, juste) est géré de manière isolée.

Cette manière de coder prépare déjà à des pratiques qu'on retrouve dans le développement professionnel : **écrire un code simple, prévisible et lisible**, où les erreurs sont gérées tôt et proprement.