TP 1 — Bulls, cows… and versions

Introduction

Dans ce TP, vous allez travailler sur le développement d'un jeu nommé « Bulls & Cows ». Votre attention devra se porter tout particulièrement sur les points suivants :

- le suivi des modifications avec Git ;
- la documentation des fonctions (docstring) ;
- les tests unitaires.

Les règles du jeu sont les suivantes.

- Chacun·e des deux joueurs et joueuses se munit d'une feuille de papier et d'un crayon. Chacun·e note son code secret en… secret donc.
- Celui-ci est composé d'un nombre de quatre chiffres.
- Chaque joueur et joueuse va ensuite s'efforcer de deviner le plus rapidement le nombre de l'autre. Pour cela, la première personne qui joue fait une proposition sous forme d'un nombre à quatre chiffres et l'autre lui répond :

  Bull pour chaque chiffre bien placé.
  Cow pour chaque chiffre mal placé.
  Rien pour les mauvais chiffres.

- On indique uniquement le *nombre* de bulls and cows, pas leur place dans le code secret.

Ainsi pour découvrir le code 2569, la personne qui cherche questionnant écrit 8267. La réponse est 1 bull (le 6 est bon et bien placé) et 1 cow (le 2 est présent mais mal placé).

Nous allons développer ce jeu en mode console où un humain doit devenir la configuration secrète choisie aléatoirement par l'ordinateur. L'objectif principal n'est pas de travailler la *conception* d'un projet, mais le suivi des versions. Nous adoptons le *workflow* Git standard, à savoir :

- La branche `master` contient uniquement du code stable, prêt à être distribué et mis en production. On ne développe *jamais* directement sur la branche `master`.
- On crée une branche `dev`, qui sera la branche de développement.
- Chaque nouvelle fonctionnalité intégrée est développée dans une sous-branche de la branche `dev`. Lorsque la fonctionnalité est opérationnelle et correctement testée, on la fusionne dans la branche `dev`.
- Lorsque le développement sur `dev` aboutit à une version du projet exploitable, on fusionne le code dans la branche `master`.
- On recommence de cette manière pendant toute la durée du développement.

Travail à rendre

Vous devez rendre l'ensemble du répertoire de votre projet *avec le dépôt git* fourni dans une archive ZIP nommée `r203_tp2_<nomDeFamille>.zip`. Comme tout projet, vous fournirez un fichier `README.md` à la racine et un fichier `.gitignore` dans votre dépôt.

Voici un exemple de découlement d'une partie :

Terminal

```
Proposition sous la forme '1234' : 4785
essai=1, : bulls=2, cows=0
--------------------------------------------------------------------------------
Proposition sous la forme '1234' : 6321
essai=2, : bulls=1, cows=0
--------------------------------------------------------------------------------
Proposition sous la forme '1234' : 4357
essai=3, : bulls=2, cows=0
--------------------------------------------------------------------------------
Proposition sous la forme '1234' : 4388
essai=4, : bulls=3, cows=0
--------------------------------------------------------------------------------
Proposition sous la forme '1234' : 4344
essai=5, : bulls=3, cows=0
--------------------------------------------------------------------------------
Proposition sous la forme '1234' : 4384
essai=6, : bulls=4, cows=0
--------------------------------------------------------------------------------
Secret trouvé après 6 essais
```

Au travail

Exercice 1 (Mise en route).

1) Créer un répertoire `R203_QDev‣TP2‣` et initialiser un dépôt git à l'intérieur.

2) Créer une branche `dev` et se placer sur cette branche.

3) Tout le code sera contenu dans le fichier `partieBC.py`. Créer ce fichier, et l'utiliser pour le premier commit.

4) Pour tester le code, on utilise un fichier source séparé, `partieBC_tests.py`, qui importe `partieBC.py` et exécute appelle les fonctions programmées pour vérifier qu'elles produisent les bons résultats.

5) Les codes de tests doivent systématiquement faire l'objets de commits spécifiques pour les identifier.

Nous allons programmer les fonctions suivantes, dont on fournit en annexe page 6 la signature et les spécifications sous la forme de docstrings.

Exercice 2 (La fonction `combinaison_secrete()`).

1) Créer une branche `nb_secret` et s'y placer.

2) Proposer un code pour la fonction `combinaison_secrete()`, en effectuant autant de commits qu'il vous semblera utile.

3) Dans le fichier `partieBC_tests.py`, appeler la fonction que vous venez de programmer pour vérifier son bon fonctionnement. Ajouter les tests dans un commit spéficique, avec le descriptif `[Tests] : fonction combinaison_secrete()`.

4) Lorsque le développement de cette fonction vous semble terminé, retourner sur la branche `dev` et y fusionner la branche `nb_secret`.

5) Supprimer la branche `nb_secret`, devenue inutile.

Exercice 3 (Saisie clavier)**.** Nous allons programmer la fonction qui lit au clavier la combinaison proposée par l'utilisateur·rice.

1) Créer une branche `saisie_clavier` et se placer sur cette branche.

2) Programmer une fonction `conversion_saisie()` qui lit une saisie clavier de quatre chiffres et renvoie la liste formée des quatre chiffres saisie. Par exemple, la saisie clavier `"1234"` doit être convertie en la liste `[1, 2, 3, 4]`.

3) Effectuer un commit lorsque vous estimez avoir terminé.

4) Votre fonction doit *vérifier* que la saisie est valide, i.e. qu'elle est bien de taille 4 et formée uniquement de chiffres. Implémenter les vérifications requises dans un commit spécifique.

Vérifier qu'une chaine de caractères est composée de chiffres

Si `chn` est une chaine de caractères, l'instruction `chn.isnumeric()` renvoie `True` si `chn` contient un nombre entier et `False` sinon.

5) Dans le fichier `partieBC_tests.py`, ajouter des tests pour vérifier le bon fonctionnement de votre fonction. Les ajouter dans un commit `[Tests] Fonction saisie_clavier()`.

6) Lorsque les tests sont concluants, basculer sur la branche `dev` et fusionner le contenu de `saisie_clavier` dans `dev`.

7) Supprimer la branche `saisie_clavier`.

Du bon usage des commits

Il est toujours délicat de savoir quand créer des commits. Des commits trop nombreux ou contenant trop peu de modifications alourdissent considérablement le dépôt et deviennent illisibles.

À l'inverse, des commits trop peu fréquents limitent les possibilités de suivre les modifications. Une bonne pratique consiste à créer un commit au moins dans les cas suivants :

- nouvelle fonction programmée ;
- ajouts de nouveaux tests ;
- fonctionnalité *en cours de développement* mais pas terminée et laissée temporairement en pause avant de basculer sur une branche différente pour développer autre chose en parallèle.

Exercice 4 (Comptage des Bulls & Cows)**.** Nous allons maintenant mettre en place la fonctionnalité qui permet, étant donnée une combinaison sous la forme d'une liste de quatre `int`, de compter le

nombre de Bulls et de Cows dans cette combinaison. Nous allons décomposer cette fonctionnalité en plusieurs fonctions :

- `histo(configuration)` reçoit une liste de quatre `int` et renvoie une liste `h` de dix `int` telle que pour tout `i` $\in [\![0, 9]\!]$, `h[i]` est le nombre d'occurrence de `i` dans `configuration`.
- `nombre_communs(configuration1, configuration2)` reçoit deux configurations de même taille en paramètre et renvoie le nombre de chiffres en commun *sans considération pour leur position*.
- `nombre_bulls_cows(configuration1, configuration2)` reçoit deux configurations de même taille et renvoie une liste à deux éléments : le nombre de chiffres en commun bien placés, et le nombre de chiffres en commun mal placés.

1) Créeé une branche `nb_bulls_cows` et placez-vous sur cette branche.

2) Programmer la fonction `histo(configuration)` dans le fichier `partieBC.py` et les tests associés dans le fichier `partieBC_tests.py`. Créez autant de commits que nécessaire. En particulier, il faut au moins un commit final lorsque la fonction `histo(configuration)` est convenablement développée et testée.

3) Procéder de même pour la fonction `nombre_communs(configuration1, configuration2)`.

4) Enfin, faire de même pour `nombre_bulls_cows(configuration1, configuration2)`

Vérification des *préconditions*

On pensera à vérifier que les conditions prérequises sont satisfaites pour les trois fonctions (longueur des listes, notamment).

5) Lorsque le développement de cette fonctionnalité est terminée, basculer sur la branche `dev` et la fusionner avec `nb_bulls_cows`.

Modification du dernier commit

Git fournit une commande qui permet de *modifier* le dernier commit. On suppose que l'on a effecté un commit ayant l'identifiant `7e9qfp3`. Après ce commit, effectue une modification mineure au fichier `partieBC.py`, que l'on souhaite intégrer au commit précédent suite à un oubli. On commence par exécuter la commande `git add partieBC.py` de manière habituelle.

Si l'on applique la commande `git commit -m "…"`, cela va créer un nouveau commit. À la place, on souhaite *intégrer* les modifications au dernier commit, identifié `7e9qfp3`. Cela se fait avec la commande `git commit --amend`.

Cette instruciton est pratique lorsque l'on a oublié d'intégrer des changemens, mais elle est à utiliser avec parcimonie car elle réduit la lisibilité de l'historique.

Exercice 5 (La fonction principale)**.** Programmer maintenant la boucle principale du programme, dans la fonction `partie_joueur_humain(nbmax_essai)`. Effectuer autant de commits que nécessaire, à chaque étape qui vous semblera utile. On travaillera sur une branche spécifique `boucle_principale`.

Lorsque le travail est terminé, fusionner le contenu dans la branche `dev`.

Le développement est maintenant terminé. Si votre code est correctement testé, vous pouvez appliquer les commandes suivnates :

```
Terminal
user@host:~/Documents/R203_QDev/TD2_BullsAndCows/ git branch
  master
* dev
user@host:~/Documents/R203_QDev/TD2_BullsAndCows/ git switch master
Basculement sur la branche
user@host:~/Documents/R203_QDev/TD2_BullsAndCows/ git merge dev
```

Pour aller plus loin

Si vous souhaitez approfondir ce TP et l'utilisation de Git, vous pouvez programmer une variante où la combinaison secrète ne peut contenir qu'une seule fois chaque chiffre (autrement dit, on interdit les doublons). Au lancement du programme, on choisit le mode de jeu (avec sous sans doublon), et ensuite la partie commence.

En l'état actuel du projet, les branches `master` et `dev` contiennent exactement le même code, stable et en bon état de fonctionnement. Pour ajouter la nouvelle fonctionnalité, on bascule sur la branche `dev`, sur laquelle on pourra créer autant de sous-branches et de commits que nécessaire. Et lorsque le développement sera terminé, on pourra de nouveau fusionner `dev` dans `master`.

Ainsi, la branche `master` contient toujours un code abouti, celui considéré comme « en production ».

Annexe

partieBC.py

```
def combinaison_secrete():
    """Renvoie un tableau de 4 entiers choisis
    aléatoirement entre 0 et 9"""

def histo(configuration):
    """Renvoie une liste du nombre d'occurences
    de chaque chiffre entre 0 et 9 dans la liste configuration"""

def nombre_bien_places(configuration1, configuration2):
    """Renvoie le nombre de chiffres identiques et
    à la même position pour deux configurations"""

def nombre_communs(configuration1, configuration2):
    """Renvoie le nombre de chiffres communs pour les deux
    configurations passées en paramètre"""

def nombre_bulls_cows(configuration1, configuration2):
    """Renvoie un tableau [nombre de bulls, nombre de cows]
    pour [nombre de chiffres bien placés, nombre de chiffres
    communs mais mal placés] pour les configurations passées
    en paramètres"""

def conversion_saisie(chaine):
    """Convertit une chaîne de caractères en tableau d'entiers"""

def partie_joueur_humain(nbmax_essai):
    """Lance une partie de Bulls and Cows : l'utilisateur.rice doit
    deviner la combinaison secrète en au plus nbmax_essai
    essais. Ses propositions sont des chaines de caractères
    comme '1234' et sont converties en tableaux d'entiers.
    """
```