Introduction à la programmation orientée objet, à la documentation du code et aux tests

Devoir maison

À rendre au plus tard le 17/03/2025 à 21h00 sur Moodle

Lisez entièrement l'énoncé avant de commencer à coder.

1 Rappels concernant le plagiat

Commençons tout d'abord par un petit rappel du règlement de scolarité :

"***Tout travail remis dans le cadre de la scolarité à l'ENSAI est un travail personnel.***"

Une conséquence directe est que si vous partagez votre devoir avec un·e autre étudiant·e, vous n'avez aucun moyen de vous assurer que le travail remis par d'autres étudiant·e·s sera personnel. **Vous n'avez donc aucune raison valide de partager votre code avec une autre personne, excepté l'autre membre de votre binôme si vous travaillez à deux**. De plus, il est souvent difficile de ne pas plagier, même involontairement, le travail d'une autre personne une fois qu'on y a accès.

Concernant les intelligences artificielles génératives, il serait malhonnête de dire qu'il est facile de toujours identifier du code généré par de tels outils. Néanmoins, il est parfois facile d'identifier du code généré par intelligence artificielle, par exemple lorsque le code utilise des bibliothèques non vues dans le cadre de ce cours ou que la documentation suit un style différent de celui vu en cours. **L'utilisation d'intelligences artificielles génératives est interdite par principe** pour les raisons suivantes :

- Dans le cas où plusieurs devoirs sont extrêmement similaires, non pas parce que vous avez copié les un·e·s sur les autres mais parce que vous avez utilisé la même intelligence artificielle avec avec les mêmes prompts, cela rentre dans le cadre du plagiat.
- Étant donné que les notions vues dans ce cours sont nouvelles pour la majorité d'entre vous, il me semble plus judicieux pour vous de vous focaliser sur l'acquis de nouvelles connaissances et de compétences. Pourquoi étudier à l'ENSAI si au final vous n'avez que peu de valeur ajoutée par rapport à une intelligence artificielle générative ?

De plus, vous avez pas mal de temps pour effectuer ce devoir maison. Si vous souhaitez aider une personne qui n'est pas dans votre binôme, il faut l'aider directement et lui fournir des explications et non lui envoyer son code. Il reste des séances de TP/TD pour poser éventuellement des questions à votre chargé de TP/TD. Vous pouvez également passer à mon bureau (si je suis présent) ou m'envoyer un email. Si vous ne souhaitez pas faire un minimum d'efforts pour effectuer ce devoir maison, il vaut mieux ne rien rendre que rendre un travail impliquant du plagiat.

Tous les codes rendus seront analysés entre eux. Vous perdrez beaucoup de points si du plagiat est détecté, à la fois pour les personnes plagiant et pour les personnes plagiées (puisqu'il est impossible de déterminer automatiquement qui est qui, et que la personne plagiée est responsable d'avoir partagée son code).

2 Objectifs

L'objectif principal de ce devoir maison est de mettre en application les concepts appris durant les cours magistraux et les travaux pratiques, notamment :

- la programmation orientée,
- la documentation du code,
- les tests, et
- le développement piloté par les tests (*test-driven development*).

3 Sujet

Dans ce devoir maison, vous allez travailler sur le rami, qui est un jeu de cartes de combinaisons. Plutôt que de plagier la page Wikipedia[1] dédiée au rami, je vous laisse la lire si vous ne connaissez pas ce jeu. **Néanmoins, comme les règles peuvent varier, vous ne devez vous baser que sur les informations fournies dans ce document**. S'il vous semble manquer des informations, merci de m'en informer par mail. De plus, l'objectif n'est pas de faire un jeu de rami entièrement fonctionnel, mais d'implémenter certaines de ses fonctionnalités.

Le reste de cette section présente les principales classes et fonctionnalités à implémenter.

3.1 Carte

Une carte est caractérisée par sa valeur et sa couleur. On considérera 52 cartes différentes, réparties en :

- 13 valeurs : 2, 3, 4, 5, 6, 7, 8, 9, 10, valet, dame, roi, as ;
- 4 couleurs : pique, cœur, carreau, trêfle.

Le joker ne sera pas modélisé.

3.2 Combinaison

Une combinaison est, comme son nom l'indique, une combinaison de plusieurs cartes. Il existe trois types de combinaisons : les brelans, les carrés et les séquences. Une combinaison est valide si et seulement si elle appartient à l'un de ces trois types. À une combinaison valide correspond un nombre de points associé.

3.3 Main

Une main correspond aux cartes qu'à un joueur dans sa main. Il est possible de piocher une carte dans la réserve, de jeter une carte dans la défausse et de poser une ou plusieurs combinaisons. La première pose a des règles spécifiques : il faut poser au moins une séquence et au moins 51 points.

3.4 Réserve

La réserve correspond au paquet de cartes dans lequel les joueurs piochent les cartes. Il est possible de distribuer les cartes aux joueurs pour commencer une partie.

3.5 Défausse

La défausse correspond aux cartes jetées par les joueurs en cours de partie. À la fin d'une partie ou si la réserve est vide en cours de partie, la défausse est mélangée puis est ajoutée à la fin de la réserve.

[1] https://fr.wikipedia.org/wiki/Rami

4 Organisation du code

4.1 Architecture

L'architecture de votre code sera la suivante :

```
src
├── _sanity_check.py
├── base.py
├── carte.py
├── combinaison.py
├── conftest.py
├── defausse.py
├── main.py
├── pyproject.toml
├── reserve.py
├── setup.cfg
├── test_base.py
├── test_carte.py
├── test_combinaison.py
├── test_defausse.py
├── test_main.py
└── test_reserve.py
```

4.2 Contenu

Les noms des modules sont explicites, mais vous trouverez ci-dessous le contenu attendu pour chaque module :

- `base.py` : implémentation de la classe `_ListeCartes`
- `carte.py` : implémentation de la classe `Carte`
- `combinaison.py` : implémentation de la classe `Combinaison`
- `defausse.py` : implémentation de la classe `Defausse`
- `main.py` : implémentation de la classe `Main`
- `reserve.py` : implémentation de la classe `Reserve`
- `test_base.py` (**fourni**): implémentation des tests pour la classe `_ListeCartes`. Vous devez obligatoirement utiliser le module fourni sans le modifier, et donc écrire du code dans le module `base.py` compatible avec les tests fournis.
- `test_carte.py` : implémentation des tests pour la classe `Carte`
- `test_combinaison.py` : implémentation des tests pour la classe `Combinaison`
- `test_defausse.py` : implémentation des tests pour la classe `Defausse`
- `test_main.py` : implémentation des tests pour la classe `Main`
- `test_reserve.py` : implémentation des tests pour la classe `Reserve`

4.3 À quoi correspondent les fichiers `pyproject.toml` et `setup.cfg` ?

Le fichier `pyproject.toml` est le fichier de configuration de référence d'un projet Python, à la fois pour *paqueter* son projet mais également pour configurer des outils externes. C'est dans ce fichier que l'on peut notamment configurer des outils tels que *Black* et *pytest*.

L'outil *Flake8* est un peu plus vieux et n'est pas compatible avec le fichier `pyproject.toml`, c'est pourquoi on passe par un autre fichier de configuration, `setup.cfg`.

4.4 À quoi correspond le module `conftest.py` ?

Comme vous l'aurez remarqué, il manque la description du module `conftest.py`. Ce module particulier a été brièvement présenté dans le deuxième cours magistral et un exemple d'utilisation sera vu dans l'exercice 2 du TP6.

Lorsque l'on écrit des tests, il est courant de vouloir définir des variables (constantes) ou des *fixtures* et les utiliser dans plusieurs modules de test. C'est impossible par défaut pour les fixtures, qui ne sont disponibles que dans le module où elles sont définies. C'est possible pour les variables, mais il faut les importer manuellement à chaque fois, ce qui alourdit le code.

Le module `conftest.py` permet de résoudre ces deux problèmes. Les *fixtures* définies dans un module `conftest.py` sont disponibles pour tout le répertoire et peuvent être utilisées sans être manuellement importées, mais il faut les rajouter en arguments des fonctions de test. Pour les variables (constantes), il suffit de les définir dans une fonction nommée `pytest_configure()`. Le module `test_base.py` illustre l'utilisation de telles variables.

Un module `conftest.py` vous est déjà fourni sur Moodle. Les 52 cartes d'un jeu de cartes ont été définies. Vous êtes libres de l'utiliser ou de ne pas l'utiliser pour l'écriture de vos tests.

4.5 À quoi correspond le module `_sanity_check.py` ?

Comme vous l'aurez remarqué, il manque également la description du module `_sanity_check.py`. Ce module particulier comporte quelques vérifications basiques, notamment sur les classes et l'existence de leurs attributs et leurs méthodes. À l'inverse, il ne vérifie pas le contenu des attributs et des méthodes. L'exécution de ce module vous permet de vérifier que vous n'avez pas effectué une erreur d'étourderie, par exemple sur la visibilité ou le nom d'un attribut ou d'une méthode.

5 Évaluation

Vous devriez l'avoir compris maintenant, mais avoir du code qui fonctionne n'implique pas que le code est de bonne qualité. Vous serez donc également évalués sur les bonnes pratiques de programmation que sont le style de code, la documentation et les tests.

5.1 Contenu à rendre

Il faut déposer, sur Moodle, votre dossier `src` avec les modules listés en section 4.1 en tant qu'archive (dossier compressé). Les formats autorisés pour votre archive sont `.zip` et `.7z`.

5.2 Critères essentiels

Respecter les contraintes fournies dans le diagramme de classes est le strict minimum. Ces contraintes sont partiellement vérifiées par le module `_sanity_check.py` : les noms et les types des attributs et des méthodes sont vérifiés, mais pas les arguments ni les sorties des méthodes. **Avant de réfléchir à l'écriture du code nécessaire pour chacune des méthodes, vous êtes fortement encouragé·e·s à écrire d'abord le *squelette* complet de chacune des classes, c'est-à-dire définir les attributs et les méthodes sans écrire le code des méthodes** (par exemple en écrivant juste `pass` dans chacune des méthodes). Les seules exceptions sont :

- les éventuels attributs d'instances, et donc
- les constructeurs (les méthodes spéciales `__init__()`), qui doivent être (partiellement) codés pour créer les attributs d'instance et pour créer les instances utilisées dans le module `_sanity_check.py`.

Une fois que ce sera fait, vous n'aurez normalement plus à vous soucier d'avoir fait une erreur d'étourderie sur le nom ou le type d'un attribut ou d'une méthode. Vérifiez tout de même, après avoir écrit le code des méthodes et avant de déposer votre devoir sur Moodle, qu'aucune erreur n'est levée en exécutant le module `_sanity_check.py`.

L'exécution du module `_sanity_check.py` ne doit pas lever d'erreurs. Dans le cas contraire, votre note ne pourra pas dépasser 10/20.

Il est également essentiel que les tests soient exécutables. Si vous avez une erreur de syntaxe dans votre code, il est fort probable que l'exécution des tests échoue. Si l'exécution des tests échoue, c'est comme si vous n'aviez écrit aucun test.

L'exécution des tests ne doit en aucun cas lever une erreur. Dans le cas contraire, vos tests ne seront pas évalués.

5.3 Style de code

Le style de votre code sera évalué avec le paquet Flake8. Pour des raisons évidentes d'équité, la version utilisée de Flake8 sera celle disponible sur les machines virtuelles, c'est-à-dire la version `6.0.0`, avec la configuration définie par le fichier `setup.cfg` (c'est-à-dire ce que vous obtenez en exécutant la commande `flake8` dans le terminal en étant dans le dossier `src`) :

```
> python -m flake8
```

ou

```
> flake8
```

Vous perdrez des points si vous avez des erreurs ou avertissements levés par Flake8. Vous êtes fortement encouragés à utiliser le paquet Python *Black* pour formater votre code, ce qui devrait également résoudre automatiquement pas mal d'erreurs ou avertissements levés par Flake8. Pour ce faire, exécutez l'une de ces commandes (dans le terminal, en étant dans le dossier `src`) :

```
> python -m black .
```

ou

```
> black .
```

5.4 Documentation

Les classes et leurs méthodes devront êtres documentées avec au minimum :

- une description courte,
- la section *Parameters* décrivant les arguments s'il y en a, et
- la section *Returns* si la méthode renvoie autre chose que `None`.

Les seules exceptions sont les méthodes spéciales qui n'ont pas à être documentées. De plus, vous n'avez pas à écrire de doctests.

Les modules de tests n'ont pas à être documentés, mais les noms des fonctions doivent indiquer explicitement ce qu'elles testent.

5.5 Tests

Les tests doivent pouvoir être exécutés avec le paquet Python `pytest`, dont la version utilisée sera celle disponible sur les machines virtuelles, c'est-à-dire la version `7.2.0`.

Les tests doivent vérifier que le résultat obtenu correspond bien au résultat attendu. En particulier, tous les cas suivants doivent être testés :

- La bonne erreur est bien levée quand elle doit être levée.
- Le résultat renvoyé correspond bien au résultat attendu.
- Les modifications effectuées correspondent bien aux modifications attendues.

Il est recommandé d'utiliser le paquet `pytest-cov` pour vous assurer que tout votre code est bien couvert par vos tests. Son utilisation est très simple : après l'avoir installé, il suffit de rajouter l'option `--cov` en exécutant la commande `pytest` dans le terminal (en étant dans le répertoire `src`) :

```
> python -m pytest --cov
```

ou

```
> pytest --cov
```

Pour voir facilement quelles lignes ne sont pas exécutées, il est recommandé d'exporter le résultat au format HTML avec la commande :

```
> python -m pytest --cov --cov-report=html
```

ou

```
> pytest --cov --cov-report=html
```

Cette commande va générer un dossier `htmlcov` avec les pages HTML d'un site internet. La page `index.html` est la page d'accueil. Double-cliquez dessus pour l'ouvrir. Vous pouvez cliquer ensuite sur chaque module pour visualiser facilement et rapidement quelles lignes de code ne sont jamais exécutées.

A Diagramme de classes

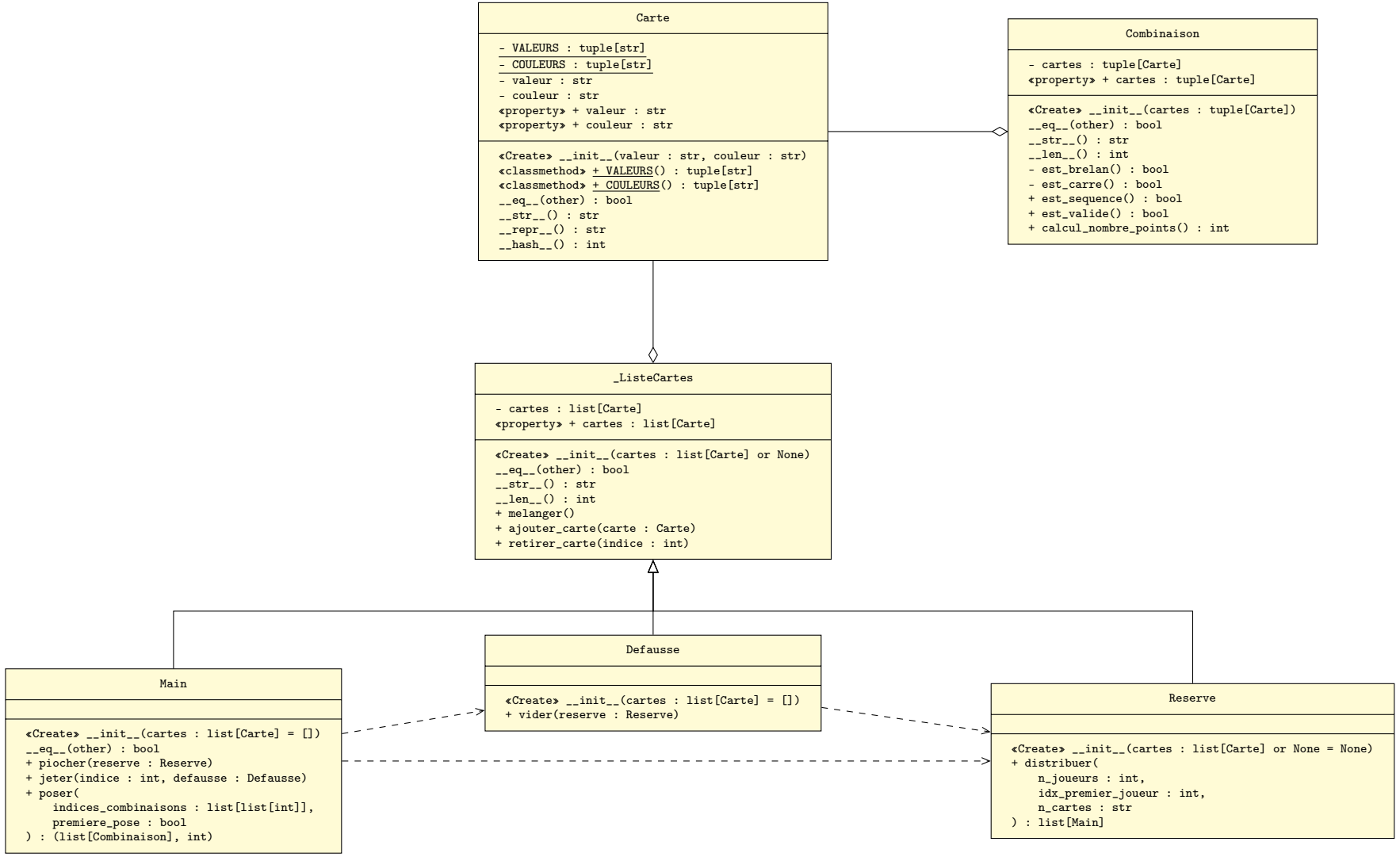

Figure 1: Diagramme de classes.

B Cahier des charges

Remarque : vous pouvez éventuellement rajouter des attributs et des méthodes. Ces éventuels ajouts doivent être parcimonieux.

B.1 Classe `Carte`

B.1.1 Attributs et méthodes de classe

Les attributs `VALEURS` et `COULEURS` sont des attributs privés de classe, indiquant les valeurs valides pour les arguments du constructeur. L'attribut `VALEURS` est un `tuple` contenant les valeurs suivantes :

```
'As', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Valet', 'Dame', 'Roi'
```

L'attribut `COULEURS` est un `tuple` contenant les valeurs suivantes :

```
'Pique', 'Carreau', 'Coeur', 'Trêfle'
```

Pour accéder aux valeurs de ces attributs de classe en dehors de la classe, on utilisera des *méthodes de classe* (voir la documentation du décorateur `@classmethod`) : la méthode de classe `VALEURS()` renverra la valeur de l'attribut de classe `VALEURS` et la méthode de classe `COULEURS()` renverra la valeur de l'attribut de classe `COULEURS`. Les `tuple` et les `str` étant immutables en Python, ces méthodes de classe n'ont pas besoin de renvoyer des copies de ces attributs de classe.

B.1.2 Attributs d'instance et propriétés

Les attributs d'instance sont `valeur` et `couleur`, indiquant respectivement la valeur et la couleur de la carte. Ces attributs sont privés. Pour permettre l'accès aux valeur de ces attributs tout en empêchant la modification ou la suppression de ces attributs, deux **propriétés** (voir les transparents du quatrième cours ou la documentation du décorateur `@property`) seront créées.

B.1.3 Méthode `__init__()`

La méthode spéciale `__init__()` de la classe `Carte` doit vérifier les valeurs des arguments et lever des erreurs si elles ne sont pas valides.

B.1.4 Méthode `__str__()`

La méthode spéciale `__str__()` de la classe `Carte` renvoie la représentation informelle d'une carte, correspondant à la manière dont les cartes sont généralement nommées. Par exemple, la représentation informelle de `Carte('As', 'Coeur')` est `'As de coeur'`, tandis que celle de `Carte('3', 'Trêfle')` est `'3 de trêfle'`.

B.1.5 Méthode `__repr__()`

La méthode spéciale `__repr__()` de la classe `Carte` renvoie la représentation officielle d'une carte. Par exemple, la représentation informelle de `Carte('As', 'Coeur')` est `"Carte('As', 'Coeur')"`, tandis que celle de `Carte('3', 'Trêfle')` est `"Carte('3', 'Trêfle')"`. Remarquez les guillements à l'intérieur de la chaîne de caractères, indiquant que les arguments sont des chaînes de caractères.

B.1.6 Méthode `__eq__()`

La méthode spéciale `__eq__()` de la classe `Carte` renvoie un booléen indiquant si la carte est égale à un autre objet. La carte est égale à l'autre objet si et seulement si :

1. l'autre objet est également une carte, et
2. leurs valeurs et leurs couleurs sont égales.

B.1.7 Méthode `__hash__()`

La méthode spéciale `__hash__()`[2] est appelée par la fonction native `hash()`. Elle est nécessaire pour rendre les cartes hachables, par exemple pour utiliser une carte en tant que clé d'un dictionnaire ou encore pour faire partie d'un ensemble.

Étant donné que deux instances de la classe `Carte` sont égales si et seulement si leurs représentations officielles sont égales, une solution (couramment utilisée) est de donner la représentation officielle en entrée de la fonction native `hash()`.

B.2 Classe `Combinaison`

B.2.1 Attribut d'instance et propriété

L'attribut d'instance est `cartes`, représentant le t-uplet de cartes de la combinaison. Cet attribut est privé. Pour permettre l'accès aux valeur de ces attributs tout en empêchant la modification ou la suppression de ces attributs, une **propriété** (voir les transparents du quatrième cours ou la documentation du décorateur `@property`) sera créée : elle doit renvoyer une **copie profonde** (voir la fonction `copy.deepcopy()`) de l'attribut privé `cartes` pour éviter la modification de la combinaison de cartes.

B.2.2 Méthode `__init__()`

La méthode spéciale `__init__()` de la classe `Combinaison` doit vérifier l'intégrité de l'argument et lever une erreur si l'intégrité n'est pas vérifiée.

B.2.3 Méthode `__eq__()`

La méthode spéciale `__eq__()` de la classe `Combinaison` renvoie un booléen indiquant si la combinaison est égale à un autre objet. Nous allons distinguer différents cas, en fonction de la nature de la combinaison.

- Si l'autre objet est une combinaison,
  - si les deux combinaisons ne contiennent aucune carte,
    - alors les deux combinaisons sont égales,
  - sinon, si la combinaison est valide et l'autre combinaison est valide,
    - alors les deux combinaisons sont égales si et seulement si elles contiennent les mêmes cartes,
  - sinon,
    - les deux combinaisons ne sont pas égales,
- sinon,
  - les deux objets ne sont pas égaux.

B.2.4 Méthode `__str__()`

La méthode spéciale `__str__()` de la classe `Combinaison` renvoie la représentation informelle d'une combinaison, correspondant au tuple de la représentation informelle de chaque carte. Par exemple, la représentation informelle de `Combinaison((Carte('As', 'Pique'), Carte('As', 'Carreau'), Carte('As', 'Trêfle')))` est `"(As de pique, As de carreau, As de trêfle)"`.

B.2.5 Méthode `__len__()`

La méthode spéciale `__len__()` de la classe `Combinaison` renvoie la longueur de la combinaison, c'est-à-dire le nombre de cartes dans la combinaison.

[2] https://docs.python.org/fr/3/reference/datamodel.html#object.__hash__

B.2.6 Méthode `est_brelan()`

La méthode privée `est_brelan()` détermine si la combinaison est un brelan. Une combinaison est un brelan si et seulement si :

1. la combinaison est constituée de trois cartes,
2. les trois cartes ont la même valeur, et
3. les trois cartes ont des couleurs toutes différentes les unes des autres.

B.2.7 Méthode `est_carre()`

La méthode privée `est_carre()` détermine si la combinaison est un carré. Une combinaison est un carré si et seulement si :

1. la combinaison est constituée de quatre cartes,
2. les quatre cartes ont la même valeur, et
3. les quatre cartes ont des couleurs toutes différentes les unes des autres.

B.2.8 Méthode `est_sequence()`

La méthode publique `est_sequence()` détermine si la combinaison est une séquence. Une combinaison est une séquence si et seulement si :

1. la combinaison est constituée d'au moins trois cartes,
2. toutes les cartes ont la même couleur, et
3. toutes les cartes se suivent.

Remarque : un as ne peut se trouver qu'au début ou à la fin d'une séquence. Par exemple, la combinaison (roi de pique, as de pique, 2 de pique) n'est pas une séquence.

B.2.9 Méthode `est_valide()`

La méthode publique `est_valide()` détermine si la combinaison est valide. Une combinaison est valide si et seulement si c'est soit un brelan, soit un carré, soit une séquence.

B.2.10 Méthode `calcule_nombre_points()`

La méthode publique `calcule_nombre_points()` calcule et renvoie le nombre de points d'une combinaison valide. Si la combinaison n'est pas valide, une erreur doit être levée. Voici les règles à utiliser pour déterminer la valeur d'une carte :

Chaque combinaison vaut un certain nombre de points. Ces points sont la somme des points attribués aux cartes qui la composent :

- *Les cartes du 2 au 10 valent leur valeur (donc de 2 à 10 points).*
- *Le valet, la dame et le roi valent 10 points chacun.*
- *L'as vaut 1 s'il est dans une séquence et précédé ou suivi d'un deux. Il vaut 11 s'il est dans une séquence et précédé ou suivi avec un roi, s'il est dans un brelan d'as, ou s'il est dans un carré d'as.*

B.3 Classe `_ListeCartes`

La classe `_ListeCartes` permet de définir les fonctionnalités communes à toutes les classes caractérisées par une liste de cartes.

B.3.1 Attribut d'instance et propriété

L'attribut d'instance est `cartes`, représentant la liste de cartes. Cet attribut est privé. Pour rendre l'accès aux valeurs de ces attributs public tout en empêchant la modification ou la suppression de ces attributs, une **propriété** (voir les transparents du quatrième cours ou la documentation du décorateur `@property`) sera créée : elle doit renvoyer une **copie profonde** (voir la fonction `copy.deepcopy()`) de l'attribut privé `cartes` pour éviter la modification de la combinaison de cartes.

B.3.2 Méthode `__init__()`

La méthode spéciale `__init__()` de la classe `_ListeCartes` vérifie l'intégrité de l'argument et doit lever une erreur si elle n'est pas valide. Si l'argument est `None`, l'attribut privé `cartes` est initialisé comme deux jeux complets de cartes. Si l'argument est une liste de cartes, l'attribut privé `cartes` est simplement égal à l'argument. Sinon, une erreur est levée.

B.3.3 Méthode `__eq__()`

La méthode spéciale `__eq__()` de la classe `_ListeCartes` renvoie un booléen indiquant si la liste de cartes est égale à un autre objet. L'instance de `_ListeCartes` est égale à l'autre objet si et seulement si :

1. l'autre objet est également une instance de `_ListeCartes`, et
2. les attributs `cartes` des deux objets sont égaux, c'est-à-dire qu'on tient compte de l'ordre des cartes ici.

B.3.4 Méthode `__str__()`

La méthode spéciale `__str__()` de la classe `_ListeCartes` renvoie la représentation informelle d'une liste de cartes, correspondant à la liste de la représentation informelle de chaque carte. Par exemple, la représentation informelle de `Combinaison((Carte('As', 'Pique'), Carte('As', 'Carreau'), Carte('As', 'Trêfle')))` est `"[As de pique, As de carreau, As de trêfle]"`.

B.3.5 Méthode `__len__()`

La méthode spéciale `__len__()`[3] est appelée par la fonction native `len()`. La méthode spéciale `__len__()` de la classe `_ListeCartes` renvoie le nombre de cartes dans l'attribut `cartes`. L'intérêt de cette méthode est de pouvoir appeler la fonction native `len()` directement sur une instance sans devoir accéder à l'attribut `cartes`.

B.3.6 Méthode `melanger()`

La méthode publique `melanger()` permet de mélanger les cartes. Pour effectuer cette opération, vous pouvez utiliser la fonction `random.shuffle()`.

B.3.7 Méthode `ajouter_carte()`

La méthode publique `ajouter_carte()` permet d'ajouter une carte à la fin de la liste de cartes. Une erreur doit être levée si l'argument n'est pas une carte.

B.3.8 Méthode `retirer_carte()`

La méthode publique `retirer_carte()` permet de retirer une carte de la liste de cartes et de la renvoyer. Une erreur doit être levée si la liste de cartes est vide. Une autre erreur doit être levée si l'indice (fourni en argument) n'est pas valide.

[3] https://docs.python.org/fr/3/reference/datamodel.html#object.__len__

B.4 Classe `Reserve`

La classe `Reserve` permet de modéliser la réserve, c'est-à-dire la liste des cartes qui sont soit distribuées aux joueur·euse·s (en début de partie) soit piochées (en cours de partie).

B.4.1 Méthode `__init__()`

La méthode spéciale `__init__()` de la classe `Reserve` appelle le constructeur de sa classe mère `_ListeCartes` pour définir l'attribut privé `cartes`.

B.4.2 Méthode `distribuer()`

La méthode publique `distribuer()` de la classe `Reserve` permet de distribuer les cartes aux joueur·euse·s en début de partie.

Des erreurs doivent être levées si :

- le nombre de joueur·euse·s n'est pas correct (le nombre de joueur·euse·s est un entier compris entre 2 et 5 inclus),
- l'indice du premier joueur ou de la première joueuse n'est pas valide (chaque joueur·euse est caractérisé·e par son indice, qui est un entier compris entre 0 (inclu) et le nombre de joueur·euse·s (exclu)),
- le nombre de cartes à distribuer n'est pas correct (les valeurs possibles sont les chaînes de caractères `"13/14"` et `"14/15"`, et la valeur par défaut est `"14/15"`),
- le nombre de cartes dans la réserve est strictement inférieur au nombre de cartes à distribuer.

La distribution des cartes se fait de la manière suivante. Les cartes sont distribuées une à une aux joueur·euse·s en répétant toujours le même cycle. Le premier joueur ou la première joueuse à recevoir une carte correspond à l'indice donné en argument. Pour le cycle, on utilisera l'ordre des naturels des entiers. La distribution s'arrête lorsque le premier joueur ou la première joueuse a reçu 14 cartes (si le nombre de carte est `"13/14"`) ou 15 cartes (si le nombre de carte est `"14/15"`).

Par exemple, s'il y a 3 joueur·euse·s et que l'indice du premier joueur ou de la première joueuse est 2 (on commence à compter à 0), alors les cartes sont distribuées de la manière suivante aux joueur·euse·s (les entiers correspondent aux indices des joueur·euse·s):

$$2 \to 0 \to 1 \to 2 \to 0 \to 1 \to 2 \to 0 \to 1 \to 2 \to 0 \to 1 \to \ldots \to 2 \to 0 \to 1 \to 2$$

La méthode renvoie une liste de mains, correspondant aux mains de chaque joueur·euse.

B.5 Classe `Defausse`

La classe `Defausse` permet de modéliser la défausse, c'est-à-dire la liste des cartes jetées.

B.5.1 Méthode `__init__()`

La méthode spéciale `__init__()` de la classe `Defausse` appelle le constructeur de sa classe mère `_ListeCartes` pour définir l'attribut privé `cartes`.

B.5.2 Méthode `vider()`

La méthode publique `vider()` de la classe `Defausse` permet de vider la défausse et de l'ajouter à la fin de la réserve après l'avoir mélangée. Autrement dit, il faut mélanger la défausse avant de la vider à la fin de la réserve.

B.6 Classe `Main`

La classe `Main` permet de modéliser la main d'un·e joueur·euse.

B.6.1 Méthode `__init__()`

La méthode spéciale `__init__()` de la classe `Main` appelle le constructeur de sa classe mère `_ListeCartes` pour définir l'attribut privé `cartes`.

B.6.2 Méthode `__eq__()`

La méthode spéciale `__eq__()` de la classe `Main` renvoie un booléen indiquant si la main est égale à un autre objet. Remarquez que nous redéfinissons la méthode spéciale `__eq__()` car elle est déjà présente dans la classe mère `_ListeCartes`.[4] La main est égale à l'autre objet si et seulement si :

1. l'autre objet est également une main, et
2. les deux mains sont égales.

Attention : l'ordre des cartes ne doit pas influer le résultat obtenu. Par exemple, si deux mains sont égales, alors elles le seront toujours après avoir mélangé chaque main. Il ne faut donc pas vérifier l'égalité des listes de cartes car l'ordre est pris en compte pour vérifier l'égalité de deux listes.

B.6.3 Méthode `piocher()`

La méthode publique `piocher()` de la classe `Main` permet de piocher la première carte de la réserve : la première carte de la réserve est enlevée de la réserve et ajoutée à la fin de la main.

B.6.4 Méthode `jeter()`

La méthode publique `jeter()` de la classe `Main` permet de jeter une carte de la main dans la défausse : la carte correspondant à l'indice est enlevée de la main et est ajoutée à la fin de la défausse.

B.6.5 Méthode `poser()`

La méthode publique `poser()` de la classe `Main` permet de poser des combinaisons de cartes. Des erreurs doivent être levées si :

- les indices ne sont pas valides,
- les indices ne sont pas uniques (on ne peut jouer une carte qu'une seule fois),
- au moins une combinaison n'est pas valide.

De plus, pour la première pose, des erreurs doivent être levées si :

- il n'y a aucune séquence parmi les combinaisons,
- la somme des points des combinaisons est strictement inférieure à 51.

Sinon, les cartes correspondant aux indices doivent être retirées de la main et la méthode renvoie la liste des combinaisons ainsi que le nombre de points posés.

[4] À noter que toute classe personnalisée hérite de la classe `object` qui a déjà une méthode spéciale `__eq__()`, mais son implémentation est particulière (elle vérifie l'identité, ce qui est bien plus fort que l'égalité) et son utilisation telle quelle est donc généralement inappropriée.