TP : Héritage

Instruction générale : hormis pour les exercices corrigés collectivement, vous ferez valider votre travail par l'enseignant.

1 Déclaration d'une classe enfant

La déclaration d'une classe enfant en Python se fait aussi à l'aide du mot clé `class`. Toutefois, on précisera après le nom de la classe enfant celui de la classe parente entre parenthèses : `class Enfant(Parent)`.

Exemple :

```
class Sorceleur(Personnage) :
    pass
```

L'intérêt de l'héritage est de **réutiliser** les attributs et méthodes de la classe parente. On peut donc les appeler pour un objet enfant de la même façon que pour un parent.

Exemple : créons un sorceleur et faisons le devenir ami avec Triss et Yennefer.

```
geralt = Sorceleur("Géralt de Riv","h")

geralt.etreAmi(triss)
geralt.etreAmi(yennefer)
```

2 Constructeur et commande `super()`

On remarque dans l'exemple précédent que même sans constructeur, il est possible de créer une instance de la classe enfant. C'est parce que celui de la classe parente `Personnage` est appelé par défaut. Toutefois, il est possible que l'on veuille modifier ou compléter le constructeur de la classe enfant ; on peut donc lui en déclarer un de la même façon que pour n'importe quelle classe : avec `__init__()`. Cela a cependant un inconvénient, c'est que le nouveau constructeur enfant vient remplacer celui du parent mais ne fait pas appel à celui-ci automatiquement. C'est là qu'intervient la commande `super()` : elle permet de faire appel aux attributs et méthodes de la classe parente dans la classe enfant. Dans le constructeur, elle est utilisée comme dans l'exemple suivant.

Exemple : ajoutons un constructeur à notre `Sorceleur`.

```
class Sorceleur(Personnage) :

    def __init__(self, nom : str, sexe : str = None) :

        """ sexe est attendu sous la forme "f" ou "h"."""

        super().__init__(nom,sexe)
        epee_acier = None
        epee_argent = None
        armure = None
```

Ce constructeur fait appel au constructeur parent en fournissant les paramètres nécessaires à ce dernier : `nom` et `sexe`. On remarquera au passage qu'il n'est pas nécessaire de lui fournir le paramètre `self`.

Exercice 1. Créer des sous-classes de la classe `Joueur` pour chaque poste : attrapeur, batteur, gardien et poursuiveur. On se contentera pour cet exercice de programmer leurs constructeurs.

3 Polymorphisme et surcharge de méthodes

3.1 Polymorphisme

Il est possible de réécrire complètement des méthodes différentes pour des classes enfants à la place de la classe parente. C'est le **polymorphisme**. La méthode de la classe enfant devient alors prioritaire sur la parente lors de l'appel de la méthode pour une instance enfant.

Exemple : reprenons notre classe `Sorceleur` et redéfinissons sa méthode `sePresenter` de façon à ce que notre sorceleur précise son métier mais plus la liste de ses amis.

```
class Sorceleur(Personnage) :

    def __init__(self, nom : str, sexe : str = None) :

        """ sexe est attendu sous la forme "f" ou "h"."""

        super().__init__(nom,sexe)
        epee_acier = None
        epee_argent = None
        armure = None
```

```
    def sePresenter(self) :

        if self.sexe.lower() == "f" :
            print(f"{self.nom}, sorceleuse.")
        else :
            print(f"{self.nom}, sorceleur.")
```

Créons une autre classe enfant de la classe `Personnage` et modifions aussi sa méthode `sePresenter`.

```
class Magicien(Personnage) :

    def sePresenter(self) :

        if self.sexe.lower() == "f" :
            print(f"{self.nom}, magicienne.")
        else :
            print(f"{self.nom}, magicien.")
```

Recréons nos instances et appelons la méthode `sePresenter`.

```
triss = Magicien("Triss Merigold","f")
yennefer = Magicien("Yennefer de Vengerberg","f")
geralt = Sorceleur("Géralt de Riv","h")

triss.sePresenter()
yennefer.sePresenter()
geralt.sePresenter()
```

3.2 Surcharge de méthodes

Il est possible de **surcharger** des méthodes grâce à la commande `super()` de la même façon que `__init__()` a été modifié.

Exemple : une autre possibilité pour modifier la méthode `sePresenter` et faire en sorte que le personnage donne son métier est la suivante :

```
class Magicien(Personnage) :

    def sePresenter(self) :

        super().sePresenter()

        if self.sexe.lower() == "f" :
            print("Je suis magicienne avant tout.")
```

```
        else :
            print("Je suis magicien avant tout.")
```

Ici, la méthode parente `sePresenter` est appelée grâce à `super()` et on y ajoute ensuite le test permettant de préciser si l'on est magicien ou magicienne. Cette façon de faire a l'avantage de conserver la méthode parente en l'appelant et allège l'implémentation.

Exercice 2. Créer une méthode `sePresenter` pour la classe `Joueur` puis surcharger ou modifier cette méthode dans les classes enfants de la classe `Joueur` afin de préciser le type de poste pour chacune d'entre elles.

Exercice 3. [Pour aller plus loin] Ajouter des méthodes et des attributs supplémentaires à vos classes. Faites interagir les instances entre elles. Le tout en fonction de vos envies.

4 Classe abstraite

Il est possible d'implémenter ce que l'on appelle des **classes abstraites**. Celles-ci ne peuvent pas être instanciées mais servent de classes parentes pour des classes enfants liées entre elles. En Python, on peut appeler ces classes abstraites grâce au module `ABC` (Absract Base Class) comme on le ferait pour une classe enfant quelconque. En effet, toutes les classes abstraites sont vues en Python comme filles de la classe `ABC`.

Exemple : considérons la classe `Monstre`, qu'est-ce qu'un monstre ? Qu'est-ce que serait une instance de monstre ? Il en existe de très nombreux types et de très différents. Le monstre est un concept abstrait. On peut donc déclarer la classe `Monstre` comme fille de la classe `ABC`.

```
from abc import ABC

class Monstre(ABC) :
    pass


class Vampire(Monstre) :
    pass


class Cocatrix(Monstre) :
    pass


class Leshen(Monstre) :
    pass
```

5 Structuration du code, modules et paquets

Comme pour n'importe quel programme, il convient de respecter les normes d'implémentation du PEP 8 : aérer son code, le documenter, utiliser des noms de variables et de fonctions explicites.

5.1 Module

À la structuration du code, il faut ajouter le découpage des programmes en plusieurs fichiers. En effet, la POO est idéale pour travailler sur des projets de grandes tailles et avoir un seul fichier pour le programme complexifie inutilement les choses. On préférera avoir un fichier pour une classe et ses sous-classes puis appeler tous les fichiers nécessaires dans un programme principal généralement nommé `main.py`.

Le fait d'avoir un fichier par classe permet d'implémenter et de maintenir plus facilement. Cela permet aussi un travail d'équipe plus aisé, un programmeur peut s'occuper d'un fichier sans interférer avec ses collègues de façon incohérente.

Un fichier unique contenant des définitions de classes est appelé **module**. On peut alors appeler le module avec la commande `import` comme on l'a fait jusqu'ici pour des modules officiels comme `math`, `random`, etc.

Exemple : on va regrouper dans un fichier `personnage.py` la classe `Personnage` et ses enfants `Magicien` et `Sorceleur`. Cela formera ainsi un module cohérent. À côté de cela, on pourra créer un autre module `monstre.py` avec la classe `Monstre` et ses enfants. On appellera le tout dans le fichier `main.py` de la façon suivante :

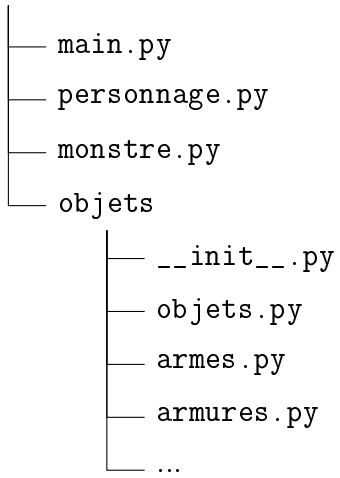
```
from personnage import Personnage, Sorceleur, Magicien
from monstre import Monstre

geralt = Sorceleur("Géralt de Riv","h")
```

Remarque : préciser les classes et fonctions à importer au lieu de toutes les importer à l'aide de `*` permet de mieux contrôler son programme et sa sécurité.

5.2 Paquet

Il est aussi possible de regrouper plusieurs modules dans des répertoires. Cela peut-être utile si l'arborescence des sous-classes devient trop grandes. Un répertoire contenant plusieurs modules est appelé **paquet**. Un paquet doit contenir un fichier `__init__.py` qui l'initialise. Celui-ci peut être laissé vide (sa seule présence suffit) ou contenir un code d'initialisation sous la forme d'une définition `__all__ = ["module_1","module_2",... ]`.

Exemple : considérons la classe `Objet` qui elle-même a pour sous-classes les classes `Arme`, `Armure`, `Potion`, `Ingrédient`, etc ; elles-mêmes ayant pour sous-classes les classes `Epee`, `Arbalete`, `Plastron`, `Gants`, etc. Regrouper toutes ces classes dans un seul fichier serait beaucoup trop lourd. Il serait plus judicieux de faire un paquet `objets` contenant un module parent `objet.py` appelé dans d'autres sous-modules `armes.py`, `armures.py`, etc. On aurait alors l'arborescence ci-contre.



Pour accéder aux modules du paquets `objets`, il suffit de préciser le nom du répertoire dans la commande `import` avec la syntaxe `paquet.module`. Ici, cela donnerait avec le module `armes.py` :

```
from objets.armes import Epee
```

Exercice 4.

1. Créer des modules `joueurs`, `balles`, `balais`, `equipes`.
2. Les appeler dans un programme principal.