Alix Dufour et Théo Bouguet

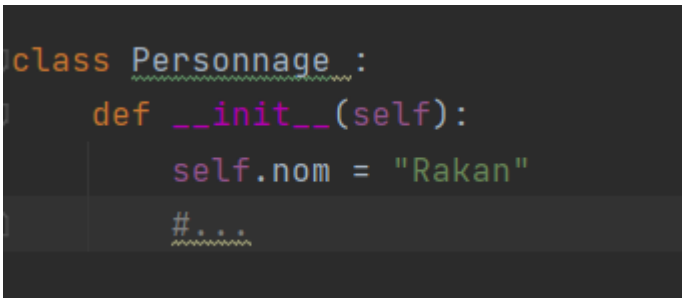
Session Automne 2022

TP2
Programmation orientée objet avancée

Question 4

Concepts de bases

Tout d'abord, voici quelques rappels sur la POO en Python. Créer une classe se fait de la manière suivantes :

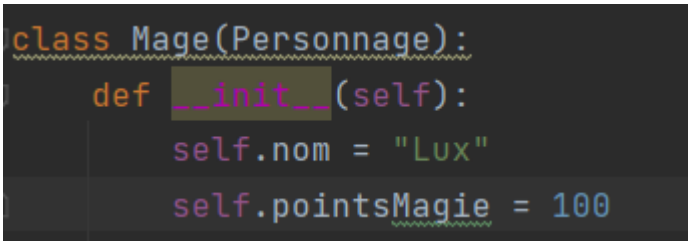


Ici, on crée une classe Personnage. On peut ensuite définir les méthodes de la classe. La méthode __init__ sur la figure ci-dessus est la méthode appelée au moment de l'instanciation d'une instance de classe (c'est le constructeur).

self.nomVariable permet de déclarer des attributs. Dans cet exemple, on a un attribut "nom" permettant de stocker le nom du personnage.

Héritage

L'héritage nous permet ensuite de créer des classes filles à partir d'une classe parente. Les classes filles vont hériter des méthodes et des attributs de la classe parente. Ici, on peut créer des classes filles afin de diviser les personnages en différents type. Par exemple, on aura des "mages", des "guerriers" et des "archer".



```python
class Guerrier(Personnage):
    def __init__(self):
        self.nom = "Garen"
        self.endurance = 100
```

```python
class Archer(Personnage):
    def __init__(self):
        self.nom = "Ashe"
        self.nombreFleches = 10
```

Pour faire hériter une classe d'une autre, il faut rajouter la classe parente lors de la déclaration de la classe fille. Par exemple class Archer(Personnage) créé une classer Archer qui hérite de Personnage.

On voit donc que les 3 classes Archer, Guerrier et Mage possède l'attribut "nom" hérité de la classe Personnage, mais possède aussi des attributs propres à elles. Pour Archer c'est "nombreFleches", pour Guerrier c'est "endurance" et pour Mage c'est "pointsMagie".

Polymorphisme, surcharge et redéfinition de méthodes.

Le polymorphisme est un concept qui permet de donner plusieurs formes à un objet, une fonction ou un attribut. On peut accomplir cela ici en redéfinissant des méthodes dans nos classe fille. Par exemple, ajoutons dans un premier temps une méthode "attaquer" dans la classe Personnage.

```python
def attaquer(self):
    print(self.nom + " attaque")
```

Ici, la méthode nous montre simplement quel personnage attaque en indiquant son nom. Mais l'on peut la redéfinir dans les 3 classes filles. En effet, chacun des types de personnages ne va pas attaquer de la même façon. Un mage va lancer un sort, un Guerrier attaque avec son épée et un archer tire une flèche.

Dans notre fichier, main, on instancie 3 personnages : un mage, un guerrier et un archer.

```
if __name__ == '__main__':
    lux = Mage()
    garen = Guerrier()
    ashe = Archer()

    lux.attaquer()
    garen.attaquer()
    ashe.attaquer()
```

Les 3 personnages appellent la méthode attaquer.

Si on lance le programme maintenant, on obtient la trace d'exécution suivante :

```
Lux attaque
Garen attaque
Ashe attaque

Process finished with exit code 0
```

Etant donné que les 3 personnages appellent la méthode de la classe parente, il est simplement affiché quel personnage attaque, mais on ne sait pas quel type d'attaque il s'agit pour chacun des personnages.

Il est cependant possible de redéfinir cette méthode dans les 3 classes filles, afin de préciser le comportement de la méthode "attaquer" pour chacun des types de personnages.

```
class Guerrier(Personnage):
    def __init__(self):
        self.nom = "Garen"
        self.endurance = 100

    def attaquer(self):
        print(self.nom + " donne un coup d'épée !")
```

```python
class Mage(Personnage):
    def __init__(self):
        self.nom = "Lux"
        self.pointsMagie = 100

    def attaquer(self):
        print(self.nom + " lance un sort !")
```

```python
class Archer(Personnage):
    def __init__(self):
        self.nom = "Ashe"
        self.nombreFleches = 10

    def attaquer(self):
        print(self.nom + " tire une flèche !")
```

Désormais, lorsque l'on exécute le fichier main, on obtient :

```
Lux lance un sort !
Garen donne un coup d'épée !
Ashe tire une flèche !

Process finished with exit code 0
```

La méthode "attaquer" possède donc plusieurs formes, pour chacune des classes filles.

En python, il n'est pas possible de surcharger nos méthodes, c'est-à-dire de définir plusieurs méthodes avec le même nom dans une classe mais ayant une signature différente. Si l'on fait ça, c'est la dernière définition qui sera prise en compe. Par exemple, dans ma classe Guerrier je veux pouvoir préciser quel personnage j'attaque en créant une nouvelle définition de la méthode "attaquer" en précisant une variable cible :

```
    def attaquer(self):
        print(self.nom + " donne un coup d'épée !")

    def attaquer(self, cible : Personnage):
        print(self.nom + " donne un coupe d'épée à " + cible.nom + " !")
```

Mais lorsque l'on exécute le fichier main, on obtient cette erreur :

```
Lux lance un sort !
Traceback (most recent call last):
  File "C:/Users/Théo/Documents/Etude_Concept_Python/main.py", line 14, in <module>
    garen.attaquer()
TypeError: attaquer() missing 1 required positional argument: 'cible'
```

Étant donné qu'il ne prend en compte que la dernière définition, il attend un argument "cible".

Il est cependant possible de surcharger les opérateurs. Par exemple, je voudrais que les opérateurs de comparaisons permettent de comparer les points de vie de chaque personnage. Je peux alors écrire dans ma classe personnage :

```
# Opérateur <
def __lt__(self, other):
    return self.pointsVie < other.pointsVie

#Opérateur <=
def __le__(self, other):
    return self.pointsVie <= other.pointsVie

# Opérateur >
def __gt__(self, other):
    return self.pointsVie > other.pointsVie
#Opérateur >=

def __ge__(self, other):
    return self.pointsVie >= other.pointsVie

def __eq__(self, other):
    return self.pointsVie == other.pointsVie
```

J’ai aussi ajouté un attribut pour garder les points de vie du personnage :

```
def __init__(self):
    self.nom = "Rakan"
    self.pointsVie = 100
```

Ensuite, dans mon fichier main, j’ai retiré des points de vie à “Lux” et j’ai fait une comparaison avec “Garen” :

```
lux.pointsVie -= 10

if(lux > garen):
    print(lux.nom + " a plus de vie que " + garen.nom)
elif(lux < garen) :
    print(lux.nom + " a moins de vie que " + garen.nom)
elif(lux == garen):
    print(lux.nom + " a autant de vie que " + garen.nom)
```

L'exécution nous donne :

```
Lux a moins de vie que Garen

Process finished with exit code 0
```

La surcharge des opérateurs a bien fonctionné.

Généricité

En POO, la généricité est le fait de pouvoir réaliser des opérations identiques sur des types différents.
En python, les variables sont typées dynamiquement, il est alors possible d'appeler des méthodes ou d'utiliser des types génériques comme les list en passant en argument n'importe quel type de variable.

Par exemple, on peut créer une liste de n'importe quel type.

```
ListeNombre = [5,7,6,9,10]
```

Dans cet exemple, on a une liste de nombre entiers.

On peut alors, si on le souhaite, appeler la méthode sort() qui permet de trier les éléments du plus petit au plus grand (ici du plus petit au plus grand nombre).

On obtient donc :

```
ListeNombre.sort()
print(ListeNombre)
```

```
C:\Users\Théo\AppData\Local\Programs\Python
[5, 6, 7, 9, 10]

Process finished with exit code 0
```

Maintenant, étant donné que les listes sont des types génériques, on peut aussi créer une liste avec d'autres types de valeur :

```
lux = Mage()
garen = Guerrier()
ashe = Archer()

lux.pointsVie -= 10
```

```
ListePersonnage = [garen, ashe, lux]
```

Ici, on crée une liste contenant des objets de type Personnage. Et grâce à la généricité, on peut aussi appeler la méthode sort() puisque l'on ne se préoccupe pas du type de données et que l'on a préalablement surcharger les opérateurs de comparaisons. En tout logique, on doit obtenir les personnage de celui qui a le moins de points de vie à celui qui a le plus de points de vie étant donné que c'est de cette manière que l'on a défini nos comparaisons précédemment.

```
ListePersonnage = [garen, ashe, lux]
print("Liste personnage avant tri :")
print(ListePersonnage)
ListePersonnage.sort()
print("Liste personnage après tri :")
print(ListePersonnage)
```

```
Liste personnage avant tri :
[<Guerrier.Guerrier object at 0x0171E658>, <Archer.Archer object at 0x0171EC10>, <Mage.Mage object at 0x01A2CBE0>]
Liste personnage après tri :
[<Mage.Mage object at 0x01A2CBE0>, <Guerrier.Guerrier object at 0x0171E658>, <Archer.Archer object at 0x0171EC10>]

Process finished with exit code 0
```

On remarque bien qu'avant le tri, les éléments sont placés selon l'ordre d'ajout dans la liste, mais qu'après le tri, Lux devient le premier élément puisque c'est le personnage qui a le moins de vie.

Modularité

En Python, il existe des modules ou bibliothèques, qui permettent de regrouper le code de différentes fonctions, qui pourront ensuite être utilisées dans d'autres programmes, après avoir été importés.
Le principe est que le développement de chacun de ces modules puisse se faire indépendamment des autres.
Cette modularité du code permet de le découper et de le rendre plus lisible, mais aussi de pouvoir plus facilement le maintenir et de réutiliser ses fonctions dans d'autres programmes.

```
from Guerrier import Guerrier
from Mage import Mage
from Archer import Archer
```

Par exemple dans main2.py on importe les modules afin d'obtenir les classes Guerrier, Mage et Archer.

Python pour l'analyse de donnée

Python est particulièrement pertinent pour plusieurs raisons :

- Tout d'abord Python est un langage assez simple, ce qui permet d'être rapide à coder, mais également d'être très clair à relire puisqu'il demande moins de lignes de code.
- Ensuite, Python dispose d'une grande communauté, et ainsi à la fois d'exemples et d'aides potentielles par d'autres utilisateurs si un développeur n'arrive pas à résoudre un problème.
- L'aspect le plus intéressant de Python pour l'analyse de données est qu'il existe de nombreuses bibliothèques efficaces qui permettent d'analyser rapidement et facilement des données telles que : NumPy, Pandas, Cython…

Pour l'exemple, nous allons utiliser le fichier "patients.csv" contenant une liste de patients, avec certaines données, ici les colonnes "Prénom", "Nom", "Naissance", et "Label". Nous avons choisi d'utiliser la bibliothèque Pandas pour les exemples, car c'est une des plus utilisées pour l'analyse de données.

Pandas permet de manipuler principalement deux types d'objets : les Series et les Dataframe qui sont assez semblables au fonctionnement d'une base de données. Assez simplement, les Series sont des colonnes et les Dataframes sont des tables composées de Series.

D'abord, on peut charger les données contenues sous différents formats, comme les CSV ou les fichiers excel. Ici, nous chargeons des données de patients écrits pour cet exemple dans un fichier csv.

```
patients = pd.read_csv(r'.\patients.csv')
```

Une seule ligne permet d'importer les données du fichier csv (ici le fichier doit être dans le même dossier que le code)

```
print(patients)
```

	Prénom	Nom	Naissance	Label
0	Alix	Dufour	2000	4
1	Théo	Bouguet	2000	4
2	Louise	Mauve	1984	5
3	Margot	Delarue	1985	2
4	Patrick	Tremblay	1999	8
5	Marguerite	Desjardins	1996	9
6	Félicie	Potdevin	2005	3
7	Pénélope	Lafonctaine	2002	2
8	Morgane	Dufour	1968	5
9	Nicolas	Damers	1978	6
10	Anthony	Guerlain	2001	7

Pandas permet alors d'afficher tout le contenu du Dataframe sous le format d'un tableau lisible.

```
print(patients.describe())
```

Ensuite la fonction describe() permet très facilement d'analyser les données numériques : ici elle analyse des données de "Naissance" et "Label", on peut ainsi voir la moyenne, l'écart-type, la médiane ou les extremums de chaque colonne très facilement.

```
          Naissance
count     11.000000
mean    1992.545455
```

Par exemple, ici on a l'information qu'il y a 11 années de naissances et que la moyenne est de 1992.

```
print(patients.sort_values(by=["Naissance"]))
```

Ensuite, toujours en une ligne on peut trier la liste, ici par année de naissance dans l'ordre croissant.

```
        Prénom          Nom  Naissance  Label
8       Morgane       Dufour       1968      5
9       Nicolas       Damers       1978      6
2        Louise        Mauve       1984      5
3        Margot      Delarue       1985      2
5    Marguerite   Desjardins       1996      9
4       Patrick     Tremblay       1999      8
0          Alix       Dufour       2000      4
1          Théo      Bouguet       2000      4
10      Anthony     Guerlain       2001      7
7      Pénélope  Lafonctaine       2002      2
6       Félicie     Potdevin       2005      3
```

On peut aussi filtrer des données : ici on n'affiche que les lignes dont le Label est égal à 4

```
print(patients[patients["Label"]==4])
```

```
  Prénom      Nom  Naissance  Label
0   Alix   Dufour       2000      4
1   Théo  Bouguet       2000      4
```

Python est aussi très pratique dans l'affichage des données sous formes de courbes. Cela se fait notamment à l'aide de la bibliothèque matplotlib. Pandas utilise justement celle-ci et permet de rapidement afficher des courbes sur les données de nos Dataframe. Par exemple, si je souhaite afficher un diagramme en bâton montrant l'occurrence d'apparition des labels, on peut écrire :

```
patients["Label"].value_counts().plot.bar();
plt.legend()
plt.xlabel("Label")
plt.ylabel("Occurence")
plt.show()
```

La 1ère ligne permet de créer le diagramme en précisant que l'on veut compter le nombre d'apparition de chaque valeur grâce à la méthode value_counts(), puis on appelle la méthode plot.bar() afin de créer le diagramme. Les lignes suivantes permettent de configurer l'affichage en ajoutant des labels et une légende et d'indiquer à Python d'afficher le graphique. Finalement, nous obtenons ceci :