Python

Programmation Orienté Objet

Comme l’indique le titre, nous allons aborder ici la Programmation Orienté Objet, souvent abrégé en « POO », en langage python.

Avant tout de chose, vous devez avoir de (bonnes) bases en programmation python, notamment les notions de variables, conditions, boucles et fonctions.

Une fois ces éléments acquis, vous pouvez vous orienter vers la programmation orienté objet. À la différence de ce vous faisiez avant, c’est-à-dire de la programmation dite « procédurale », la POO est beaucoup utilisé dans le milieu professionnel et permet de réaliser des choses plus « flexibles » et complets. Il permet une sorte de « liberté d’expression », entre autres.

La plupart des langages de programmation sont orienté objets de base, comme le Java et le C++, et Python l’est également.

Vous ne vous en êtes pas rendu compte, mais en réalité, en codant de façon procédurale, vous utilisiez déjà de l’objet. Ce que vous allez faire par la suite, c’est de « créer » de l’objet.

Sommaire

Une petite explication

Je vous ai dis que vous aviez déjà utilisé de l'objet auparavant sans vraiment savoir ce que c'est. Où me direz-vous ? Voyons un exemple tout de suite.

Créons une liste : L = []

Jusque là, rien de nouveau. Pour ajouter un élément dans cette liste, il faut utiliser la fonction **append()**. Par exemple : L.append(« 92 »)

append() est appelé « fonction » du fait de sa ressemblance aux fonctions définies par **def** ou **lambda**, mais son vrai nom est « méthode ». Ajouté à cela, le type « **list** », qui défini toute liste, est en réalité une « classe ». Donc nous pouvons ainsi dire : **append** est une méthode de la classe **list**. Un objet est donc une classe, grosso-modo. Ne creusez pas plus loin, les explications viendront après.

Ainsi, ce que vous allez principalement faire, à l'issu de ces diapositives, c'est de créer vous-même vos classes.

Un affichage formaté

Non, toujours pas. Avant d'aborder le vrai sujet, voici un outil permettant d'afficher des éléments de manière plus ... mieux.

Il s'agit de la méthode/fonction **format()** de la classe **str** (string, pour les chaines de caractères). Elle est très simple d'utilisation. Un exemple tout d'abord (ou deux) :

```
date = 19
print("Nous sommes le {} octobre {}".format(date, 2018))

T = "Il est actuellement {0}h{1}"
print(T.format(18, 34))
```

```
Nous sommes le 19 octobre 2018
Il est actuellement 18h34
```

La fonction **format()** (appelons cela comme fonction pour le moment jusqu'à ce qu'on aborde les méthodes, mais c'est une méthode), s'utilise pour le « formatage » d'une chaine de caractères. Il permet en quelque sorte d'arranger la chaine. Ici, il nous servira seulement pour l'affichage avec la fonction **print()** (qui est aussi une méthode de la classe **str**, gardez cela dans un coin).

Comme pour **append()**, il s'utilise à la suite d'une chaine de caractères, comme dans l'exemple, dans une variable ou telle quelle (telle qu'elle ?). Syntaxe :

une_chaine_de_caracteres.**format**(...)

Vous avez du remarquer que dans la chaine de caractères il y avait des paires d'accolades { }, et que dans les parenthèses de la fonction **format**, il y avait des variables. Ce que va faire cette fonction, c'est placer les éléments entre parenthèses à la place de chaque paire de { }, tout simplement, et ce, dans l'ordre dans lequel les variables sont placées dans les ( ). Notez qu'il y a des chiffres dans les { } du deuxième exemple. Ils sont facultatifs.

Hormis des chiffres, les { } peuvent accueillir autre chose. Re-exemple :

```
print("{sujet} {negation1}{verbe} {negation2}".format(verbe="arrive", negation2="pas", sujet="L'hiver", negation1="n'"))
```

```
L'hiver n'arrive pas
```

Qu'ai-je donc fait ? J'ai affecté les éléments dans les ( ) à une variable, variables que j'ai placé entre les { } dans la chaine de caractère principale. De ce fait l'ordre n'a plus d'importance.

L'avantage d'utiliser **format()**, c'est qu'il n'y a plus besoin de changer les types de chaque élément. Prenez l'une des deux chaines de la diapo 5 : les entiers (**int**) n'ont pas été convertis en chaine de caractères **(str)**. Or pour concaténer un **int** et un **str**, il aurait fallu changer le type des **int**. Pour deux ou trois variables oui ce n'est pas vraiment utile, mais pour 5, voir plus, ça l'est.

Les classes

Les classes / Création de classe

Enfin vient le moment véridique, j'ai nommé les classes !

En deux trois mots, une **classe** est un élément ou un ensemble d'élément contenus dans une structure fonctionnelle et plus ou moins ordonnée (parce que c'est pas toujours ordonné, hmmm).

Pour illustrer cela en images :

Des vêtements pliés, plus ou moins rangés : une liste

Un dressing : une classe

Cela vous interpelle t-il ?

La liste n'est qu'un entassement de variables, alors qu'une classe, c'est claaaaaaaaaaaasse ! Mais c'est surtout un ensemble fonctionnel ! Qui dit fonctionnel dit fonction ! Oui car dans une classe il y a des fonctions, appelées « méthodes », qu'on ne pas va pas encore voir tout de suite car ... Il faut créer la classe avant de la remplir ! Comment ? En utilisant le mot clé « **class** ». La syntaxe est la suivante :

```
class NomDeLaClasse(object):
        # quelque chose
        # quelque chose
        # ...
```

- **class** : toute création de classe débute par le mot clé « class »
- **NomDeLaClasse** : c'est le nom que vous donnerez à votre classe. La convention veut que si le nom est composé de plusieurs mots, la première lettre de chaque mot est mise en majuscule. Pas de ponctuation, ni d'espaces !
- **object** : il s'agit de la classe **objet**. Notez juste qu'il faut placer cela ici. Les explications viendront dans la partie « héritage ».
- « **:** » : à la fin de la ligne, comme pour les fonctions
- Toujours comme pour les fonctions, les instructions doivent être indentées

Chose faite, il faut maintenant y mettre des instructions. Contrairement aux fonctions qui retournent une valeur avec **return**, les classes ne retournent pas de valeur. Vous ne pouvez donc pas laisser votre classe telle qu'écrite comme précédemment, vous devez forcément lui passer quelque chose.

S'il est possible de ne mettre qu'un **return** seul dans une fonction, il existe un mot clé que l'on peut utiliser pour les classes si on ne veut rien y mettre et la remplir plus tard. Oui il est possible de modifier une classe après l'avoir créé.

Il s'agit du mot clé « **pass** ». Je ne vais pas l'expliquer, il est très facile de deviner sa fonction rien qu'en la voyant : il permet de « passer ». Vous pouvez retrouver son détail dans la partie erreurs des diapositives sur les bases en python.

Donc, comme pour **return**, on fait comme suit :

```
class NomDeLaClasse(object):
        pass
```

L' élément créé sera alors de type « class »

```
<class '__main__.NomDeLaClasse'>
```

Félicitations !

Vous venez de créer votre première classe.

Maintenant votre tâche est de la remplir, en commençant par « l’initialiser ». Et pour cela, il vous faut connaître les « méthodes » !

Les classes / Méthodes et attributs

Définition

Qu’est-ce qu’une méthode ?
Tout simplement, c’est une fonction de classe. Voilà. Merci. Au-revoir.

Plus sérieusement, une méthode est une fonction définie dans la classe elle-même et permet de faire toute sorte de manipulation sur les « attributs » de la classe.

Tu as dis « attributs » ?
Les attributs sont des variables propres à la classe. Comme pour les fonctions, les variables créées dans les fonctions ne sont pas accessibles depuis l’extérieur. Elles sont « locales ».

Créer/définir une méthode

Créer une méthode revient à créer une fonction. Pour cela, il faut utiliser le mot clé **def**.

Lors de la création d'une classe, il faut l'initialiser (ou non), mais en général oui. Pour cela nous allons utiliser ce qu'on appelle une méthode « spéciale ». Il s'agit d'une méthode ... spéciale. Mettez cela dans un coin, on y reviendra plus tard.

Quand une classe est créée, une méthode est appelée automatiquement pour l'initialisation (si elle existe). Il s'agit de la méthode **__init__()** (2 underscores ou tirets-bas de chaque côté). Cette fonction prend obligatoirement et au minimum un paramètre : le paramètre **self**.

Illustration puis explication juste après :

```
class UneClasse(object):
        def __init__(self):
```

Paramètre **self**

Le mot clé **self** est LE mot clé dans la POO en python. Vous verrez que vous allez utiliser ce mot partout dans votre classe. Littéralement, il signifie « soi/soi-même » ou « à moi ». En Java, il peut s'apparenter à « **this** ». Lorsque vous allez créer une fonction ou une variable, il faut précisez que cette fonction/variable « appartient » à la classe et qu'elle travaille avec des variables de la classe. C'est grâce à l'emploi de **self** que vous allez le signifier.

Donc toute méthode qui requiert un accès à une variable « dans la classe » se doit d'avoir le mot clé **self**.

Il est également possible d'avoir des méthodes sans ce mot clé qui n'affectent pas la classe.

Attributs

Les variables interne à une classe sont appelés « attributs ».

Pour créer un attribut, il est nécessaire de placer **self** devant ce dernier pour les raison évoquées dans la diapositive précédente.

Un premier exemple :

```
class Point(object):
        def __init__(self):
                self.x = 1
                self.y = 3
```

Votre première classe « complète » ! Il s'agit d'une classe **Point** comportant deux attributs qui sont les coordonnées du point x et y.

Maintenant que votre classe est créée, vous allez vouloir accéder à ce qu'il y a dedans. Illustration :

Premièrement, j'ai affecté ma classe dans une variable **P**.

Pour accéder aux attributs, il suffit d'appeler l'attribut par son nom tout en le précédent du nom de la auquel j'ai affecté la classe, soit **P**.

Enfin, un affiche formaté avec la méthode **format** de la classe « **str** ».

```
class Point(object):
        def __init__(self):
                self.x = 1
                self.y = 3

P = Point()
x, y = P.x, P.y

print("x : {}, y : {}".format(x, y))
```

```
x : 1, y : 3
```

Il est également possible d'accéder aux attributs sans affectation préalable de la classe, mais ce n'est pas conseillé.

```
print(Point().x)
```

```
1
```

Initialisation dynamique

La méthode **__init__** prend au minimum un paramètre, **self**. Mais il peut également prendre d'autres paramètres, permettant une initialisation dynamique, et non pas par défaut.

Dans l'exemple ci-après, deux manières d'initialisation dynamique vous sont présentées.

```
class Point(object):
        def __init__(self, x, y):
                self.x = x
                self.y = y

class PointBis(object):
        def __init__(self, x=0, y=0):
                self.x = x
                self.y = y

P = Point(6, 4)
Pbis1 = PointBis()
Pbis2 = PointBis(2, 3)

print("x : {}, y : {}".format(P.x, P.y))
print("x : {}, y : {}".format(Pbis1.x, Pbis1.y))
print("x : {}, y : {}".format(Pbis2.x, Pbis2.y))
```

```
x : 6, y : 4
x : 0, y : 0
x : 2, y : 3
```

Lors de l'affection de la classe dans la variable **P**, comme pour les fonctions, il suffit de préciser les valeurs que l'on souhaite affecter aux attributs, dans les ( ) de la classe.

Pour la classe **PointBis**, des valeurs par défauts sont déjà insérées dans la méthode **__init__**. Si aucun paramètre n'est passé lors de l'appel de la classe (variable Pbis1), les valeurs (0, 0) seront affectées aux attributs. Sinon ils prendront les valeurs spécifiées (variable Pbis2).

Note : le fait que le nom des variables soient les « mêmes » (affecter x dans self.x) n'a pas d'influence

À vous de jouer : votre première méthode

Vous savez maintenant comment créer une méthode d'initialisation. Votre premier exercice sera d'écrire la méthode permettant d'afficher de façon formaté (comme dans les exemples) les variables x et y sans appeler la fonction **print** en dehors de la classe.

Tips :

- Si vous travaillez sur les attributs, n'oubliez pas le mot clé **self**
- Votre méthode ne doit pas comporter d'underscores/tirets-bas
- Pas de **return**, c'est une fonction d'affichage avec **print**
- Pour utiliser votre méthode, rappelez-vous de l'exemple avec les listes

Cela ne devrait pas vous prendre plus d'une minute si vous avez bien tout lu jusqu'à maintenant. La réponse dans la prochaine diapositive.

Tic tac tic tac tic tac ... Vous avez fini ? La réponse par là

```python
class Point(object):
        def __init__(self, x, y):
                self.x = x
                self.y = y

        def affiche(self):
                print("x : {}, y : {}".format(self.x, self.y))

P = Point(1, 2)
P.affiche()
```

```
x : 1, y : 2
```

Le point délicat ici était d'appeler votre méthode. Il fallait se rappeler de la méthode **append** de la classe **list**. C'est la même syntaxe. Tout le reste n'avait rien de compliqué. Si vous n'avez pas réussi, relisez les diapositives précédentes et réessayez.

Les méthodes : une liberté d'expression

L'avantage dans l'utilisation des classes est que vous pouvez créer toute sorte de méthode afin de réaliser (presque) tout ce que vous voulez. À vous d'adapter cela en fonction de vos besoins.

Il a été précisé qu'à la diapositive 12, les classes ne comportent pas de valeur de retour, soit de **return**. Mais rien de nous empêche d'en mettre à la fin de vos méthodes. Après tout, ce sont des fonctions !

```
class Point(object):
        def __init__(self, x, y):
                self.x = x
                self.y = y

        def retourne(self):
                return "La fonction retourne {} et {}".format(self.x, self.y)

P = Point(5, 6)
print(P.retourne())
```

```
La fonction retourne 5 et 6
```

Exercice : énoncé

Vous venez de marcher plusieurs kilomètres dans le désert et vous arrivez à une station essence. Coup de chance, vous voyez une voiture abandonné qui semblent être en état de rouler. Vous jetez un œil à l'intérieur : les clés sont sur le contact mais il n'y a plus d'essence, pas de bol. Vous ne possédez pas un sous sur vous. Un coup d'œil sur la banquette arrière vous permet de trouver une porte monnaie « magique » et la carte grise de la voiture.

Le porte monnaie vous donnera une somme déterminée aléatoirement une seule fois, avec un minimum de 0 euro et un maximum de 200 euros. Vous décidez de tenter votre chance et d'utiliser ce que vous allez obtenir pour acheter de l'essence et partir du désert pour rejoindre une ville. Vous regardez une carte de la région située dans la station et vous notez que la ville la plus proche se trouve à 30 kilomètres. En regardant la carte grise du véhicule, vous constatez que la voiture consomme 1,60 litres au kilomètre. Le prix de l'essence de la station est de 2,30 euros le litre.

Exercice : votre travail

La somme que vous allez obtenir aléatoirement sera stockée dans une variable, en dehors de la classe.

Votre voiture sera de classe **Voiture**. Vous l'initialiserez avec un attribut 0, représentant l'essence disponible dans la voiture, et un attribut booléen à **False**, représentant le possible démarrage ou non de la voiture. La voiture démarre si elle a au moins 1 litre d'essence.

Votre voiture dispose de trois méthodes (en plus de celle qui initialise) :

- Une première méthode vous permettant de remplir votre véhicule d'essence
- Une deuxième permettant de démarrer votre voiture. Si la voiture démarre, affichez « Vroum ! ». Le cas échéant, affichez « Tonnerre de Brest ! »
- Une troisième méthode vous permettant « de rouler » si la voiture est démarrée, sinon levez une erreur indiquant que vous devez d'abord démarrer la voiture. Si le plein effectué vous permet d'atteindre la ville, affichez « Super ! ». Sinon affichez « Mille millions de mille sabords !!! » ainsi que la distance que vous allez pouvoir parcourir.

Modélisez votre aventure dans la station !

Les classes / Docstring

Lorsque vous réalisez un programme, en utilisant de la programmation objet ou non, il est (parfois) nécessaire de le commenter afin de comprendre plus facilement à quoi il sert. Pour des petits programmes de 5 lignes peut être pas, mais vous serez surement amenés à réaliser des programmes de 100 lignes, voir plus. Cela sert d'une part à vous, si par exemple vous voulez reprendre un programme que vous avez réalisé il y a longtemps et que vous avez oublié ce qu'il faisait, mais également à d'autres personnes si vous avez décidé de partager votre travail.

La **docstring** est principalement intégré à la suite d'une définition d'une classe ou d'une fonction, indenté pour qu'il soit situé dans le bloc des instructions. C'est un texte écrit un toutes lettres expliquant comment fonctionne telle ou telle chose ou à quoi elle sert. Il est entouré de part et d'autre par 3 guillemets, de 2 façons possibles :

```
class Point(object):
        """Classe Point contenant les coordonnees x et y"""
        def __init__(self):
                """
                Methode d'initialisation de la classe Point
                Coordonnees x et y
                """
                self.x = 1
                self.y = 2
```

En utilisant la fonction **help()**, la docstring sera affichée ainsi que les méthodes définies dans la classe.

```
help(Point)
```

```
Help on class Point in module __main__:

class Point(builtins.object)
 |  Classe Point contenant les coordonnees x et y
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Methode d'initialisation de la class Point
 |      Coordonnees x et y
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Les classes / Propriétés

Retour sur les attributs

Avant de passer aux propriétés, un petit retour sur les attributs est nécessaire. Lorsque vous avez créé votre classe, et initialisé des attributs, il est possible de les modifier en dehors de la classe comme suit :

```
class Point(object):
        def __init__(self, x, y):
                self.x = x
                self.y = y

P = Point(0, 1)
print("x : {}, y : {}".format(P.x, P.y))

P.x = 14
P.y = 6
print("x : {}, y : {}".format(P.x, P.y))
```

```
x : 0, y : 1
x : 14, y : 6
```

Conventionnellement, et dans tous les langages de programmation orienté objet, pour accéder et modifier un attribut, il est nécessaire de passer par ce qu'on appelle des « accesseurs » et « des mutateurs ». Il s'agit du principe de « l'encapsulation ».

À partir de là, quelques éléments vont changer, notamment la façon dont vous aller nommer vos attributs.

- Un attribut que vous nommerez comme fait depuis le début comme par exemple **self.x**, pourra être obtenu et modifiable sans accesseur et mutateur.

- Un attribut dont l'accès et la modification ne peut se faire qu'à l'intérieur d'un variable doit être précédé d'un underscore/tiret-bas « _ » comme ceci : **self._x**.

- Même principe pour les méthodes que vous allez créer pour les accesseurs et mutateurs. Leur nom doit être précédé d'un « _ » (**def _methode():** ...)

Encapsulation

Leur nom en dit long sur leur signification.

Un accesseur permet d'accéder à un attribut, et un mutateur permet de la modifier. En gros, ce sont des méthodes travaillant sur les attributs.

Encore par convention, un accesseur est défini par le mot « **get** », qui signifie « avoir », « obtenir ». Un mutateur lui est défini par « **set** » qui signifie « mettre », « modifier ».

Voyons un exemple tout de suite.

```
class Point(object):
        def __init__(self, x, y):
                self._x = x
                self._y = y

        def _get_x(self):
                return self._x

        def _set_x(self, valeur):
                self._x = valeur
```

Dans cette exemple, j'ai créé un accesseur et un mutateur pour l'attribut x. Pour l'attribut y, ce sera le même principe.

Comment les utiliser ? Vu comme ceci, la première chose qui nous vient à l'esprit c'est de faire **P._get_x()** pour obtenir x si notre variable s'appelle P. Mais rappelez vous ! Une méthode précédé d'un « _ » ne peut pas être utilisé de cette manière.

Alors comment faire ?
C'est là qu'intervient les « propriétés ».

Propriétés

Une propriété est un attribut regroupant 4 méthodes :

- Une méthode accesseur
- Une méthode mutateur
- Une méthode suppresseur
- Une méthode aide

Les 2 premiers sont les plus importants, et les 2 derniers sont ... moins importants.

Créer une propriété revient à « encapsuler », ou « regrouper » ces 4 méthodes en un attribut. Pour cela, nous allons utiliser le mot clé **property**. La syntaxe est la suivante :

un_attribut = **property**(methode_get, methode_set, methode_del, methode_help)

ou bien

un_attribut = **property**(methode_get, methode_set)

Exemple

```
class Point(object):
        def __init__(self, x, y):
                self._x = x
                self._y = y

        def _get_x(self):
                return self._x

        def _set_x(self, valeur):
                self._x = valeur

        x = property(_get_x, _set_x)

P = Point(0, 0)
P.x = 2

print("x : {}".format(P.x))
```

```
x : 2
```

Que s’est-il passé ?

L’attribut **_x** n’est pas accessible depuis l’extérieur de la classe. Pour cela, les méthodes **_get_x** et **_set_x**, qui sont respectivement l’accesseur et le mutateur de l’attribut **_x** ont été transformés en propriétés et mis dans l’attribut x. x est devenu une propriété de l’attribut **_x**.

En tapant **P.x**, le programme va aller chercher la propriété x, qui elle va utiliser l’accesseur **_get_x** pour nous renvoyer la valeur de x, soit **self._x**. Et c’est en faisant **P.x = 2** que le mutateur **_set_x** a été utilisé.

Finalement, les propriétés ne sont qu’un genre de pare-feu ou déguisement permettant un accès plus conventionnel de nos attributs. Mais il est nécessaire de les créer, même si cela peut paraître anodin.

Les méthode de suppression et d’aide partent sur le même principe. Il ne seront pas détaillés ici.

Note : vous n’êtes pas dans l’obligation d’utiliser d’inclure **get** et **set** dans le nom de vos accesseur et mutateur, vous pouvez les nommer comme vous le voulez.

Les classes / Attributs de classe

Cette partie reste encore assez floue pour moi, donc je vais rapidement dessus. Je reviendrai mettre à jour cette partie lorsque j'en saurai un peu plus.

Les attributs que nous avons vu jusque là sont en réalité des « attributs objet ». Il faut maintenant différencier cet attribut avec un autre type d'attribut qui sont les « attributs de classe ».

Ces attributs dits de classe sont des attributs qui sont créées et initialisés en dehors des méthodes, dans la classe.

```
class Point(object):
        forme = "rond"

        def __init__(self):
                self.x = 1
```

forme est un attribut de classe, tandis que **x** est un attribut objet. L'avantage de ce type d'attribut c'est qu'il reste le même à chaque fois qu'une classe **point** se créée, tandis que les attributs objets peuvent varier. Les attributs de classe ne sont destinés à être modifiés car ils caractérisent la classe en quelque sorte. Mais il est possible de les modifier.

```
class Point(object):
        forme = "rond"

        def __init__(self, x):
                self.x = x

P = Point(2)
print(P.forme)
P.forme = "carre"
print(P.forme)
```

```
rond
carre
```

L’accès se fait de la même manière que les attributs objet.

```
P = Point(2)
print("avant modification de la forme de P :", P.forme)
P.forme = "carre"
print("apres modification de la forme de P :", P.forme)

P1 = Point(3)
print("forme de la classe Point non affecté pour un nouveau point :", P1.forme)

Point.forme = "carre"

P2 = Point(4)
print("apres modification de la forme de la classe Point :", P2.forme)
```

```
avant modification de la forme de P : rond
apres modification de la forme de P : carre
forme de la classe Point non affecté pour un nouveau point : rond
apres modification de la forme de la classe Point : carre
```

Regardez bien cet exemple.
Après création de la variable P, l'attribut **forme** de P a été modifié. Une variable P1 a ensuite été créée, mais l'attribut **forme** est celle définie par la classe au début, soit « rond ». Ensuite, une modification de l'attribut **forme** au niveau de la classe a été effectuée. Les variables créées à la suite ont été affectées par la modification. C'est une des propriétés des attributs de classes.

Je préfère m’arrêter ici pour cette partie avant de vous raconter n’importe quoi. Il y a pleins de concepts que je n’ai pas compris parfaitement et demande plus de travail.

Je cite notamment les méthodes de classe et les méthodes statiques. Dans une prochaine mise à jour, cette partie sera complété, mais en attendant, je vous laisse la liberté de vous informer par vos propres moyens.

Exercice

Un nouvel exercice assez amusant pour mettre en pratique ce que vous avez appris jusque là. Le but ici sera de modéliser une mitraillette virtuelle.

Dans les fichiers additionnels, téléchargez les fichiers « fire.wav » et reload.wav ». Ces deux fichiers sont des bandes son modélisant respectivement le bruit de coups de feu et la recharge de l'arme. Il va vous falloir également télécharger la bibliothèque **pygame**.

Le fichier machinegun.py contient les lignes de code nécessaire pour lire ces fichiers. Exécuter le fichier tel quel va lire le fichier « fire.wav ». Une initialisation du programme est réalisé avec **pygame.mixer.init()**, à ne pas modifier. Le groupe suivant constitué de 3 lignes permet de lire la bande son une fois. À répéter donc si vous voulez lire plusieurs la bande son.

Votre travail sera donc de créer une classe « mitraillette » avec des méthodes permettant de recharger et de tirer.

Vous disposez de 200 balles, mais vous ne pouvez que charger 50 balles à la fois dans votre arme. Il y en a déjà 50 dans le chargeur (mais vous pouvez commencer avec 0).

- Pour tirer une fois, il faudra appuyer sur le bouton « entrée » de votre clavier. Ce qui aura pour effet de jouer la bande son « fire.wav ». En « un tir », l'arme tire entre 1 et 20 balles. À la fin de chaque tir, vous afficherez dans le terminal le nombre de balles restant dans le chargeur.

- Le nombre de balles tirés en un tir sera déterminé par une variable aléatoire. Si ce nombre est supérieur au nombre de balles qu'il reste dans le chargeur, toutes les balles sont tirées.

- Vous ne pouvez pas tirer si votre chargeur est vide. Un message affiché dans le terminal vous indiquera de recharger votre arme si vous essayez de tirer.

- Pour recharger votre arme, tapez « r » dans le terminal, suivi d'une validation avec le bouton « entrée ». Le fichier « reload.wav » sera alors joué, et vous rechargerez alors jusqu'à 50 balles, s'il vous en reste.

- Pour ranger votre arme (arrêter le programme), tapez « e », puis validez avec « entrée ». Si vous n'avez plus de balles ni dans le chargeur ni en stock, vous rangez votre arme.

Tips

- La ligne de commande **sys.exit**() du module **sys** vous permet d'arrêter le programme. Ce module est par défaut déjà installé sur votre ordinateur.
- Évitez de condenser votre code en une méthode, et faites en plusieurs. Chacune ayant une action précise.
- La partie la plus complexe est la recharge de l'arme. Exploitez toutes les situations.
- N'oubliez pas le principe de l'encapsulation.
- Si vous n'arrivez pas à installer le module **pygame**, ou qu'il ne fonctionne pas, faites sans.

Les classes / Méthodes spéciales

Une méthode spéciale diffère d'une méthode « normale » de plusieurs éléments :

- Elle est reconnaissable part la présence de deux « _ » underscores/tirets-bas de part et d'autre de son nom
- Elle est exécuté automatiquement lorsqu'une certaine action est réalisée
- Il faut ... la faire (mais c'est pas vraiment une différence ...)

Nous avons déjà commencé à utiliser une méthode spéciale sans la voir en détails. Il s'agit de la méthode **__init__** (diapo 16).

__init__

Cette méthode est exécuté lorsque qu'une classe est utilisée. Comme son nom l'indique, il permet d'initialiser un objet, en créant des variables à l'intérieur, ou non. C'est un « constructeur ».

Elle prend au moins un paramètre qui est **self**, et peut en prendre d'autre qui serviront à l'initialisation de variables.

Cette méthode étant déjà détaillé à la diapo 16 et suivantes, nous nous arrêteront ici pour cette méthode.

Il existe de nombreuses autres méthodes spéciales. La liste étant longue, je ne vais vous ne présentez quelques uns.

__str__ et __repr__

Ces deux méthodes permettent un affiche de l'objet, mais ne fonctionne pas pareillement.

- La méthode **__str__** est appelée lorsque vous utilisez **print** sur l'objet. Vous pouvez formater à votre guise l'affichage que vous souhaitez obtenir. C'est également cette méthode qui est appelée lorsque vous utilisez la fonction **str()** pour convertir l'objet en chaine de caractères.

- La méthode **__repr__** est appelée lorsque vous tapez l'objet directement dans le terminal, sans passer par un script. Mais vous pouvez passer par le script en utilisant la fonction **repr()**.

Si la méthode **__str__** n'est pas définie mais que vous utilisez **print**, c'est la méthode **__repr__** qui est appelée.

Si aucune des deux n'est définie, lorsque vous souhaiterez afficher votre objet, il s'affichera son adresse mémoire.

```
class Point(object):
        def __init__(self, x, y):
            self.x = x
            self.y = y

P = Point(1, 2)
print(P)
```

```
<__main__.Point object at 0x009611D0>
```

```
class Point(object):
        def __init__(self, x, y):
            self.x = x
            self.y = y

        def __str__(self):
                return "x : {}, y : {}".format(self.x, self.y)

P = Point(1, 2)
print(P)
print(str(P))
```

```
x : 1, y : 2
x : 1, y : 2
```

```
class Point(object):
        def __init__(self, x, y):
            self.x = x
            self.y = y

        def __repr__(self):
                return "y : {}, x : {}".format(self.y, self.x)

P = Point(1, 2)
print(P)
print(repr(P))
```

```
y : 2, x : 1
y : 2, x : 1
```

```
>>> class Point(object):
...     def __init__(self, x, y):
...             self.x = x
...             self.y = y
...     def __repr__(self):
...             return "y : {}, x : {}".format(self.y, self.x)
...
>>> P = Point(1, 2)
>>> P
y : 2, x : 1
>>> repr(P)
'y : 2, x : 1'
```

Opérateurs

Je vous vous lister ici plusieurs méthodes spéciales concernant les opérateurs. Cette liste n'est pas exhaustive. Il en existe d'autres mais, ceux là sont les principaux.

opérateur	méthode	opérateur	méthode
+	__add__	<	__lt__
-	__sub__	<=	__le__
*	__mul__	>	__gt__
/	__truediv__	>=	__ge__
%	__mod__	==	__eq__
**	__pow__	!=	__ne__
& (and)	__and__	| (or)	__or__

Exemple : __add__

```
class Panier(object):
        def __init__(self):
                self.nb_pommes = 4
                self.nb_bananes = 6

        def __add__(self, val):
                PP = Panier()
                PP.nb_pommes += val[0]
                PP.nb_bananes += val[1]
                return PP

        def __str__(self):
                return "pommes : {}\nbananes : {}".format(self.nb_pommes, self.nb_bananes)

P = Panier()
P2 = P+[30, 17]
print(P2)
```

```
pommes : 34
bananes : 23
```

Explications

Dans cet exemple, j'ai créé un **Panier** avec 4 pommes et 6 bananes. Par la suite, j'ai voulu ajouter 30 pommes et 17 bananes à mon panier. Mon **Panier** ne présentait pas de fonction ajout, donc j'en ai créé une. Toutefois, à la sortie de ma fonction, je ne souhaite pas qu'obtenir les nouveaux nombres de fruits que mon panier contient. Pour cela j'aurai écris :

```
def __add__(self, val):
        self.nb_pommes += val[0]
        self.nb_bananes += val[1]
        return self.nb_pommes, self.nb_bananes
```

Je voudrais que ma fonction me retourne un **Panier** avec le nombre de fruits mis à jour à l'intérieur. Pour cela, j'ai créé un nouveau **Panier** PP dans mon **Panier** P. C'est dans ce nouveau panier PP que j'ai mis à jour le nombre de fruits. Je retourne enfin PP.

J'ai créé une nouvelle variable P2, contenant le panier avec les valeurs mises à jour, mais j'aurai pu la remettre dans P.

Exemple : __lt__ (lower than)

```
class Personne(object):
        def __init__(self, prenom, taille):
                self.prenom = prenom
                self.taille = taille

        def __lt__(self, P):
                if self.taille < P.taille :
                        return True
                else :
                        return False

P = Personne("Bernard", 170)
P2 = Personne("Gregoire", 176)

print(P < P2)
```

```
True
```

Exemple : __and__

```
class Exemple(object):
        def __init__(self, element):
                self.element = element

        def __and__(self, ex):
                return [self.element, ex.element]

E1 = Exemple("balle")
E2 = Exemple("ballon")
print(E1 & E2)
```

```
['balle', 'ballon']
```

__getattr__ et __setattr__

- La méthode **__getattr__** est appelée lorsque vous essayez d'accéder à un attribut qui n'existe pas. Les instructions qui figurent dans la méthode seront exécutées. Vous pouvez par exemple afficher un message d'erreur ou bien rediriger vers un attribut existant.

- La méthode **__setattr__** est appelée lorsque vous essayer de créer un attribut et d'y affecter un élément. Si cette méthode n'est pas définie dans la classe, ce sera la méthode de la classe **object** qui sera appelé (classe définissant toute classe).

En effet, si dans votre méthode vous écrivez :

```
def __setattr__(self, attr, val):
        self.attr = val
```

Alors, la méthode **__setattr__** de l'objet sera appelé, qui va appeler la méthode **__setattr__** de l'objet ... Et se transformera en boucle infini. C'est pour cela qu'il faudra utiliser la méthode **__setattr__** de la classe **object**. Mais pourquoi la redéfinir me direz-vous ? Et bien, en créant un attribut, peut être souhaiterez vous afficher un message comme quoi votre attribut est bien créé ou qu'une autre action soit exécutée par exemple.

Exemple : __getattr__

```
class Exemple(object):
        def __init__(self, element):
                self.element = element

        def __getattr__(self, nom):
                return "L'attribut {} n'existe pas".format(nom)

E = Exemple("voiture")
print(E.elemen)
```

```
L'attribut elemen n'existe pas
```

```
        def __getattr__(self, nom):
                return self.element

E = Exemple("voiture")
print(E.elemen)
```

```
voiture
```

Exemple : __setattr__

```
class Exemple(object):
        def __setattr__(self, attr, val):
                self.attr = val

E = Exemple()
E.element = "voiture"
```

```
    self.attr = val
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

```
class Exemple(object):
        def __setattr__(self, attr, val):
                print("L'attribut {} a bien ete cree".format(attr))
                return object.__setattr__(self, attr, val)

E = Exemple()
E.element = "voiture"
```

```
L'attribut element a bien ete cree
```

__delattr__

Méthode appelée lorsque vous souhaitez supprimer un attribut.

```
class Exemple(object):
        def __init__(self):
                self.objet = "Diamant"

        def __delattr__(self, attribut):
                raise AttributeError("Suppression impossible")


E = Exemple()
del E.objet
```

```
    raise AttributeError("Suppression impossible")
AttributeError: Suppression impossible
```

Bonus : __setattr__

Un bon exemple dans l'utilisation de la méthode **__setattr__** : empêcher la modification d'un attribut.

```
class Exemple(object):
        def __init__(self):
                self.objet = "Diamant"

        def __setattr__(self, attribut, valeur):
                raise AttributeError("Modification impossible")

E = Exemple()
E.objet = "Je vous l'emprunte"
```

```
    raise AttributeError("Modification impossible")
AttributeError: Modification impossible
```

__getitem__ et __setitem__

Toutes les méthodes dont le nom possède le mot « **-attr-** » ne concernaient que les attributs de type **int**, **float**, **str**, ... Il existe d'autres méthodes qui concernent les types **list**, **tuple**, **dict**, ... Ces types sont appelés « conteneurs » car ils « contiennent » des variables.

- La méthode **__getitem__** est appelé lorsque l'on souhaite accéder à un variable dans une liste par exemple en tapant : **une_liste[i]**
- La méthode **__setitem__** est appelé lorsque l'on souhaite affecter une variable dans une liste à une certaine position : **une_liste[i] = « x »**

Exemple

```
class Exemple(object):
        def __init__(self):
                self.objet = [1, 2, 3, 4, 5]

        def __getitem__(self, indice):
                return self.objet[indice]

        def __setitem__(self, indice, valeur):
                self.objet[indice] = valeur

E = Exemple()
print(E[0])
E[0] = 0
print(E[0])
```

```
1
0
```

Il existe encore une multitude de méthodes spéciales. Je vais m’arrêter ici pour cette partie sinon on ne va pas s’arrêter. Le but étant de vous donner les bases pour que vous continuiez à apprendre de façon autonome par la suite.

Bonus : __name__

Vous avez surement déjà du voir cet élément après de nombreuses lignes de code et à la fin d'un fichier python :

```
if __name__ == '__main__' :
        # instructions
        # instructions
        # ...
```

La langage python ne possède pas de fonction principale « main », contrairement aux autres. Ainsi, dans un fichier python, tout le contenu est passé en revu et exécuté.

La condition **if** illustrée ci-dessus permet d'éviter cela. **__name__** est une variable présente dans tous les fichiers python et contient le nom du fichier qui est importé, s'il est importé. Si vous vous situez dans le fichier principal (où vous avez importé tous vos modules et fichiers), la variable **__name__** contiendra alors '**__main__**' indiquant que vous êtes dans le fichier principal. La programme sera exécuté si et seulement si il est appelé par son nom, et non importé.

Fichier principal

```
import module

print("Hello !")
print(__name__)
```

Fichier module.py

```
print("Affichage provenant du fichier {}.py".format(__name__))
```

Terminal

```
Affichage provenant du fichier module.py
Hello !
__main__
```

Fichier module.py

```
if __name__ == '__main__':
        print("Affichage provenant du fichier {}.py".format(__name__))
```

Terminal

```
Hello !
__main__
```

Héritage

Le concept de l'héritage est ce pourquoi l'utilisation des classes est intéressante. En effet, ce concept permet de faire interagir les classes entre elles. On ne va pas y aller par plusieurs chemin, le plus simple reste le meilleur.

L'héritage ici permet de créer des classes à partir d'autres classes tout en « héritant » des propriétés de la classe à partir duquel la nouvelle classe est créée. C'est-à-dire qu'une classe créée sur la base d'une autre pourra utiliser ses méthodes et attributs sans les redéfinir.

Prenons un exemple.
Soit une classe « Cylindre », ayant deux attributs : une pour la hauteur et une pour le rayon. À partir de cette classe, je vais pouvoir créer d'autres classes (un objet cylindrique, vous l'aurez deviné), comme par exemple une bouteille, un tonneau, un baril, une canette, ... Ces objets présentent un point commun : ils sont sous forme cylindrique, et donc partagent les mêmes propriétés qui sont d'avoir une hauteur et un rayon. Ces deux valeurs sont définies dans la classe dite « mère », la classe « Cylindre ». Les classes nouvellement créées sont appelées classes « filles », créées sur la base de la classe mère. Les attributs hauteur et rayon existant déjà, on ne va pas les recréer mais les modifier. Par la suite, l'ajout d'autres attributs est également possible pour « spécialiser » nos classes filles.

Mais ...

En réalité, depuis le début, vos classes héritaient déjà d'une autre. Il s'agit de la classe « object » (diapo 11). `class NomDeLaClasse(object):`

En effet, la classe « object » contient tous les éléments nécessaires à la création d'une classe (méthodes, attributs, méthodes spéciales, etc.).

Cependant, si vous utilisez une version supérieure ou égale à Python 3, il n'est pas nécessaire de faire cette manipulation. Si vous utilisez une version antérieure à la 3 (2.7 par exemple), il vous faut faire l'héritage (recommandé). Le fait d'hériter de la classe « object » permet d'utiliser les outils les plus récents et les plus à jour dans l'utilisation des classes, tandis que créer une classe sans et avec une version inférieure à python 3, dispose des anciens outils.

Si vous ne connaissez par la version que vous utilisez, tapez « python » dans le terminal. Les informations sur la version de python que vous utilisez s'afficheront, suivi de deux ou trois chevrons à la suite « > » indiquant que vous vous trouvez dans l'environnement python. Un coup d'œil dans la liste des programmes installés fonctionne également.

```
(venv) C:\Users\Romain\Documents\python_test>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NVIDIA Pilote audio HD : 1.3.37.1	NVIDIA Corporation	10/05/2018		1.3.37.1
NVIDIA Pilote du contrôleur 3D Vision 390.41	NVIDIA Corporation	10/05/2018		390.41
NVIDIA Pilote graphique 397.64	NVIDIA Corporation	10/05/2018		397.64
Pilote de contrôleur d'hôte extensible Intel® USB 3.0	Intel Corporation	18/02/2017	18,4 Mo	3.0.0.16
Python 3.7.0 (32-bit)	Python Software Foundation	15/10/2018	92,1 Mo	3.7.150.0
Python Launcher	Python Software Foundation	15/10/2018	1,77 Mo	3.7.6386.0
Realtek High Definition Audio Driver	Realtek Semiconductor Corp.	22/02/2017	527 Mo	6.0.1.8036
RomStation	RomStation	20/05/2018	677 Mo	
Samsung USB Driver for Mobile Phones	Samsung Electronics Co., Ltd.	05/03/2017	24,6 Mo	1.5.63.0
Skype™ 7.3	Skype Technologies S.A.	21/02/2017	49,2 Mo	7.3.101

Héritage / Héritage simple

Comment cela fonctionne ?

Pour faire hériter une classe mère A dans une classe B fille, il faut faire comme suit :

```
class A(object):
        pass

class B(A):
        pass
```

Lors de la création d'une classe, comme vous savez très bien le faire maintenant, il suffit de passer, à la suite du nom de la nouvelle classe, et entre parenthèses, le nom de la classe mère (il faut qu'elle existe, sinon cela ne sert à rien). Et paf ! ~~Ça fait des choca~~ ... Votre classe B hérite de toutes les fonctionnalités de la classe A (mais également des fonctionnalités que la classe A hérite, ici « object »). B hérite de A, qui A hérite de object. Donc B hérite aussi de object.

Un premier exemple

```
class A(object):
        def __init__(self):
                self.a = 1
                self.b = 2

        def affiche(self):
                print(self.a, self.b)

class B(A):
        pass

P = B()
print(P.a, P.b)
P.affiche()
```

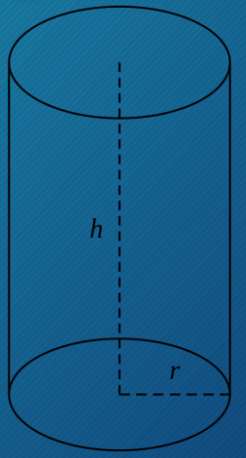
```
1 2
1 2
```

Il est possible d'accéder aux méthodes et attributs de la classe mère simplement par le biais de la classe fille comme fait dans cet exemple.

La classe fille B ici ne contient rien. Il est donc aisé d'accéder aux éléments de la classe mère. Toutefois, l'ajout de certains éléments dans la classe fille peut en modifier. C'est le concept de l'overriding, qui vous sera présenté plus loin.

Illustration

Un deuxième exemple avec notre classe « Cylindre ». On va admettre que les objets que je vais créer ont une forme de cylindre de révolution, bien que ce soit faux en réalité (une bouteille ayant une extrémité conique).




```
class Cylindre(object):
        def __init__(self, hauteur, rayon):
                self.hauteur = hauteur
                self.rayon = rayon
```

Voici une partie de notre classe Cylindre, mais on ne va pas s'arrêter là. Nous allons maintenant ajouter deux méthodes : une calculant l'aire du cylindre, et une pour le volume. Essayez donc de les faire vous-même, la réponse est à la diapo suivante.

Aire d'un cylindre : $2\pi rh + 2\pi r^2$

Volume d'un cylindre : $\pi r^2 h$

r : rayon

h : hauteur

```
class Cylindre(object):
        def __init__(self, hauteur, rayon):
                self.hauteur = hauteur
                self.rayon = rayon

        def aire(self):
                A = 2*3.14*self.rayon*self.hauteur+2*3.14*(self.rayon**2)
                return A

        def volume(self):
                V = 3.14*(self.rayon**2)*self.hauteur
                return V
```

Voici une façon de faire, mais vous pouvez la faire autrement. L'avantage dans l'utilisation de l'héritage dans cet exemple est qu'il ne sera pas nécessaire de réécrire les méthodes aire et volume dans le cas ou je vais créer une classe Bouteille se basant sur Cylindre. Comment allez-vous créer votre classe bouteille de hauteur et rayon défini dynamiquement par vous, l'utilisateur ?

La première chose qui va venir à votre esprit est ceci :

```
class Bouteille(Cylindre):
        def __init__(self, hauteur, rayon):
                self.hauteur = hauteur
                self.rayon = rayon
```

Overriding

Je vous répondrais oui c'est correct mais pas totalement. En effet, vous venez de réécrire la méthode **__init__** de la classe mère. Pour cet exemple, deux lignes ne représentent pas grand-chose, mais pour des méthodes plus développées si.

Que venez-vous de faire ? Vous venez de réaliser un « overriding », ou en français un « écrasement ». En réécrivant la méthode **__init__**,vous avez écrasé/effacé/remplacé la méthode de la classe mère par celle de la classe fille. Les deux méthodes étant ici quasi identiques, la différence ne se voit pas.

L'exemple à la diapo suivante sera plus clair.

Dans cet exemple, il y a trois classes :

- une première classe **UnObjet**
- une deuxième **UnTruc**, héritant de **UnObjet**
- une troisième **UneChose**, héritant lui aussi de **UnObjet**

Dans la classe **UnTruc**, la méthode affiche vient écraser celle de la méthode mère **UnObjet**. Ceci est observé lors de l'affichage « Bonjour » à la place du « Hello ». On a ainsi overridé la méthode affiche.

Dans la classe **UneChose**, il n'y a pas eu d'overriding. L'affichage « Hello » le prouve.

```
class UnObjet(object):
        def affiche(self):
                print("Hello")

class UnTruc(UnObjet):
        def affiche(self):
                print("Bonjour")

class UneChose(UnObjet):
        pass

A = UnObjet()
B = UnTruc()
C = UneChose()

A.affiche()
B.affiche()
C.affiche()
```

```
Hello
Bonjour
Hello
```

Comment éviter d'overrider une méthode ? Revenons à notre classe Bouteille. Pour éviter l'overriding, il suffit d'appeler la méthode **__init__** de la classe mère dans celle de la classe fille comme ceci :

```
class Bouteille(Cylindre):
        def __init__(self, hauteur, rayon):
                Cylindre.__init__(self, hauteur, rayon)

T = Bouteille(6, 4)

print(T.rayon)
print(T.hauteur)
```

```
4
6
```

Si vous ne voyez pas l'intérêt de cette manipulation, sachez qu'elle est utile, comme pour le concept des propriétés (diapo 30 et suivantes).

L'héritage permet de créer une classe sur la base d'une autre, mais pas seulement. La classe nouvellement créée peut contenir des éléments qui lui sont propre. D'une classe cylindre, peut découler une multitude d'objets. Et chacun de ces objets ont des caractéristiques particulières. Exemple avec deux classes :

```
class Bouteille(Cylindre):
        def __init__(self, hauteur, rayon):
                Cylindre.__init__(self, hauteur, rayon)
                self.matiere = "plastique"
                self.contenu = "eau gazeuse"

class RouleauDePapierToilette(Cylindre):
        def __init__(self, hauteur, rayon):
                Cylindre.__init__(self, hauteur, rayon)
                self.poids = 200
                self.utilisation = "hygiène"

T = Bouteille(6, 4)
print("Bouteille en {} contenant de l'{}".format(T.matiere, T.contenu))

R = RouleauDePapierToilette(4, 2)
print("Rouleau de papier toilette pesant {}g et ayant une fonction d'{}".format(R.poids, R.utilisation))
```

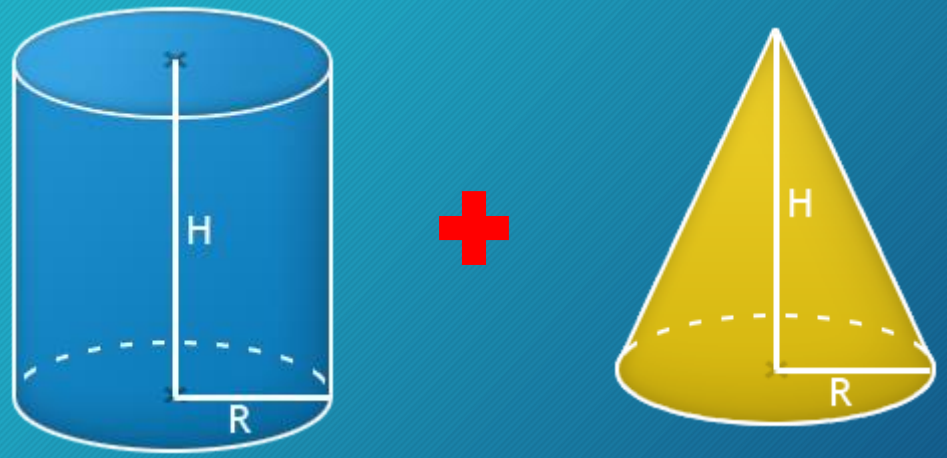
```
Bouteille en plastique contenant de l'eau gazeuse
Rouleau de papier toilette pesant 200g et ayant une fonction d'hygiène
```

Je viens de découvrir qu'il existe une page wikipédia sur le papier toilette.
Si vous êtes intéressés : https://fr.wikipedia.org/wiki/Papier_toilette

Super

Oui tout cela est super, mais ce n'est pas le but de cette diapo, de qualifier mes sublimes diapositives de « supers ». Le but est de vous introduire le mot clé **super** qui permet d'utiliser une méthode de la classe mère dans une classe fille.

Pour vous illustrer le fonctionnement de **super**, nous allons calculer le volume de notre Bouteille de manière plus précise. En effet, nous allons diviser l'objet en deux parties : un cylindre de révolution et un cône de révolution. La volume total de la Bouteille sera la somme de ces deux volumes.



Pour ce faire, nous disposons déjà de la méthode de calcul du volume d'un cylindre dans la classe mère. Le volume du cône reprend exactement la même formule, si ce n'est qu'elle est divisée par 3 à la fin. Nous allons ainsi réutiliser la méthode de la classe mère plutôt que de la réécrire.

Volume d'un cône : $\frac{volume\ cylindre}{3} = \frac{\pi r^2 h}{3}$

Reprenons notre classe Cylindre légèrement modifiée. Nous nous intéressons qu'au volume. J'ai donc enlevé la méthode de calcul de l'aire pour plus de lisibilité. Notez que j'ai remplacé **self.hauteur** par **h** pour une utilisation plus flexible de la méthode volume.

```
class Cylindre(object):
        def __init__(self, hauteur, rayon):
                self.hauteur = hauteur
                self.rayon = rayon

        def volume(self, h):
                V = 3.14*(self.rayon**2)*h
                return V
```

Comment ma Bouteille sera-t-elle dimensionnée ? La hauteur de la partie cylindre sera égale à 3/4 de la valeur de l'attribut hauteur, et les 1/4 restant iront pour le cône.

J'ai par la suite, réécrit la méthode volume. Il s'agit d'un overriding mais pas totalement. Regardons la de plus près. Elle est divisé en 2 parties : la variable V contiendra le volume de la partie cylindrique, tandis que la variable C, celui de la partie conique.

```
class Bouteille(Cylindre):
        def __init__(self, hauteur, rayon):
                Cylindre.__init__(self, hauteur-hauteur//4, rayon)
                self.hauteur_cone = hauteur//4

        def volume(self):
                V = super(Bouteille, self).volume(self.hauteur)
                C = super().volume(self.hauteur_cone)/3
                return C+V
```

Pour appeler une méthode de la classe mère, la syntaxe est la suivante :

super(ClasseFille, self).**methode_mere**(paramètres, ...)

ou bien

super().**methode_mere**(paramètres, ...) - à partir de Python 3

Si vous omettez le mot **super** lors de la réécriture de la méthode volume dans la classe fille, celle-ci va réappeler la méthode volume de la classe fille et ainsi de suite (récursion). Une boucle infini s'en suit. Le fait d'appeler une méthode de la classe mère dans une classe fille évite de tout réécrire. Dans l'exemple présenté, le calcul du volume se fait simplement en une ligne. La réécrire est rapide mais, sachant qu'elle existe déjà, il suffit de la rappeler.

```
T = Bouteille(8, 2)
print(T.volume())
```

```
83.73333333333333
```

Héritage / Héritage multiple

Jusque là, nous n'avons vu que l'héritage simple : une classe hérite d'une autre, et une seule. Mais une classe peut en hériter de plusieurs, d'où l'héritage multiple, et c'est là que cela devient intéressant. En effet, une classe peut se baser sur deux, trois, quatre classes, voir plus.

À titre d'exemple, vous pouvez vous servir d'une bouteille comme d'un objet pour transporter de l'eau mais également comme d'une arme et taper avec, ou jouer au foot ...

Comment ? Telle est la question. Comme ceci !

```
class ClasseFille(ClasseMere1, ClasseMere2, ...)
```

Tout simplement comme un héritage simple, en y ajoutant autant de classes mères que vous voulez.

Exemple !!!!!!!!!

```
class ClasseMere(object):
        def affiche1(self):
                print("Je suis ta mere !")

class ClassePere(object):
        def affiche2(self):
                print("Je suis ton pere !")

class ClasseFille(ClasseMere, ClassePere):
        pass

F = ClasseFille()
F.affiche1()
F.affiche2()
```

```
Je suis ta mere !
Je suis ton pere !
```

Remarque

Dans le cas où dans les classes mères, il y a des méthodes qui ont le même nom, seule la première, dans l'ordre dans lequel les classes sont placées dans les ( ) de la classe fille, est considérée.

```
class ClasseMere(object):
        def affiche(self):
                print("Je suis ta mere !")

class ClassePere(object):
        def affiche(self):
                print("Je suis ton pere !")

class ClasseFille(ClasseMere, ClassePere):
        pass

F = ClasseFille()
F.affiche()
```

```
Je suis ta mere !
```

```
class ClasseMere(object):
        def affiche(self):
                print("Je suis ta mere !")

class ClassePere(object):
        def affiche(self):
                print("Je suis ton pere !")

class ClasseFille(ClassePere, ClasseMere):
        pass

F = ClasseFille()
F.affiche()
```

```
Je suis ton pere !
```

Exercice

Nous/vous allons/allez créer une classe **ActionRecipient** qui va nous permettre soit de remplir la Bouteille, soit de la vider. Quelques consignes :

- la classe Bouteille va hériter de Cylindre et de ActionRecipient
- on ne peut pas remplir plus que la bouteille ne peut contenir
- on ne peut pas vider la bouteille si elle est déjà vide
- si la Bouteille peut encore accueillir 3 unités de liquide mais qu'on en rajoute 5, la Bouteille est remplie de 3 unités
- un message indique que la Bouteille est pleine si on essaye à nouveau de la remplir
- un message indique que la Bouteille est vide si on essaye de la vider alors qu'elle est déjà vide

À vous de jouer !

Tips

- Pour ma part j'ai ajouté que deux attributs dans la méthode **__init__** de la classe Bouteille ainsi qu'une méthode d'affichage. Le reste ne change pas
- La classe **ActionRecipient** doit contenir une méthode remplir() et une vider()
- Cette classe va agir sur les deux attributs nouvellement créées dans la classe Bouteille
- Il s'agit de modification d'attributs. Il n'y a pas besoin de retourner de variable
- Le fichier bouteille.py contient le squelette du programme dans les fichiers additionnels. À vous de le remplir/modifier.

À suivre …