TNSI	Représentation des données Programmation objet	Cours – **Programmation objet**

Objectifs :

- ⇨ Découvrir un nouveau paradigme de programmation
- ⇨ Connaître les principes de base de la programmation objet
- ⇨ Connaître le vocabulaire de base associé à la programmation objet
- ⇨ Programmer et utiliser des classes dans le cadre de programmes simples

L'objet de ce cours est de présenter la Programmation Orientée Objet (POO), ses concepts et son vocabulaire. La POO est un paradigme de programmation (une façon d'écrire un programme) très utilisé et présent dans de nombreux langages de programmation (C++ est la version POO du langage C par exemple).

I – Paradigmes de programmation

On appelle **paradigme de programmation** une *façon* d'écrire un programme.

Pour l'instant, le seul paradigme que nous avons vu est le paradigme **impératif** : le programme décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.
C'est historiquement la première approche et la seule supportée directement par les microprocesseurs. C'est aussi la plus utilisée.

On a depuis développé d'autres approches qui sont plus adaptées à certains problèmes. Cette année, nous ne parlerons que des paradigmes **fonctionnel** et **objet** qui sont les seuls au programme.

Le **paradigme fonctionnel** ramène le calcul à l'évaluation de fonctions éventuellement imbriquées.

L'objectif ici est d'une part de simplifier l'écriture du programme qui gagne en lisibilité (mais pas nécessairement en facilité de compréhension) et surtout d'autre part de se débarrasser des effets de bord.

Rappel :

Par exemple la procédure `print()` a l'effet de bord de modifier l'affichage. Une fonction de tri en place d'un tableau a pour effet de bord de modifier le tableau.

II – Objets et Classes

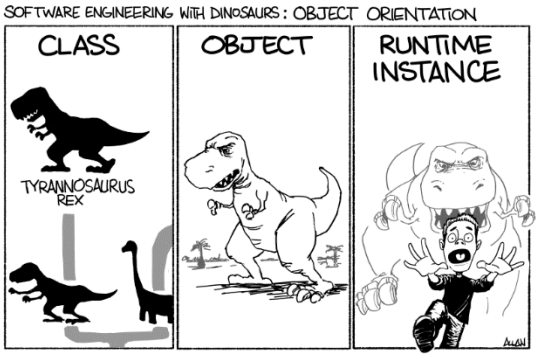
1) Classes et instances

En programmation objet, un **objet** est une entité informatique embarquant de l'information et capable de la manipuler.

Une **classe** correspond à un type d'objet.

Une **instance d'une classe** A est ..... .

Exemple :
On pourrait définir la classe Animal qui permet de représenter des animaux.
Milou, Rantanplan, Simba et Jolly jumper sont des instances de la classe Animal.



En python, on déclare une classe avec le mot-clé `class`.
Et on crée une instance de cette classe en appelant son constructeur (fonction qui a le même nom que la classe).

```
class Animal:
    pass          # Classe vide

m = Animal()      # Milou
r = Animal()      # Rantanplan
```

Remarques :

- Les noms de classe commencent toujours par une majuscule (norme implicite mais pas obligation du langage).
- `m = Animal()` crée un objet de type `Animal`. On dit qu'on *instancie* la classe (on crée une instance de celle-ci).

2) Attributs

Pour l'instant ces objets existent, mais n'ont aucun intérêt car rien ne distingue une instance d'une autre. Il faudrait que chaque objet puisse avoir ses propres caractéristiques / paramètres. C'est le rôle des attributs des objets.

Un **attribut** est ..... .

Le *nom* de cet objet est le même pour toutes les instances de la classe, mais sa *valeur* est propre à chaque instance.

On parle aussi de ..... ou de ..... d'un objet.

Exemple :
On peut définir l'attribut `nom` pour la classe `Animal`.
`m.nom` vaut `"Milou"`, `r.nom` vaut `"Rantanplan"`.

On remarque qu'on accède à la valeur d'un attribut avec l'opérateur « `.` » appliqué à la variable objet.

On pourrait définir le nom d'un animal avec `j.nom = "Jolly-Jumper"`, mais on verra plus tard (voir « encapsulation ») que c'est une mauvaise pratique. On préfèrera définir la valeur des attributs d'un objet au moment où on le crée. Pour cela il faut écrire une fonction constructeur spécifique.
Voyons comment faire cela en python :

```
class Animal:
    def __init__(self, nom, famille, nombre_de_pattes="non défini"):
        self.nom = nom
        self.famille = famille
        self.pattes = nombre_de_pattes

m = Animal("Milou", "mammifère", nombre_de_pattes=4)
k = Animal("Kaa", "reptile", nombre_de_pattes=0)
t = Animal("Titi", "oiseau", nombre_de_pattes=2)
```

Remarques :

- La fonction **constructeur**[1] s'appelle toujours « `__init__` » (avec deux sous-tirets (tirets du 8)) au début et à la fin du nom de fonction) c'est habituellement la première fonction de la classe.
- Le premier argument de la fonction `__init__` est `self`. « `self` » est une variable spéciale qui désigne l'objet lui-même et qui permettra donc d'identifier l'instance à laquelle on a affaire (et par exemple de ne pas mettre la valeur `"Rantanplan"` au champ « `nom` » de `m`, mais bien à celui de `r`.
- Une classe peut avoir autant d'attribut qu'on le souhaite, ici on en a fixé trois (`nom`, `famille` et `nombre_de_pattes`).

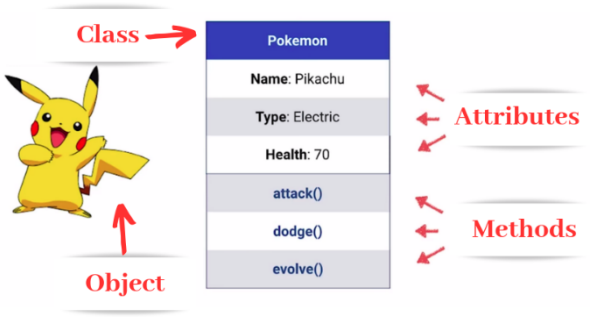
3) Les objets comme structures de données

On peut bien évidemment utiliser les objets comme structure de données composites permettant de stocker dans un seul objet plusieurs informations de nature variées.
Est-ce bien utile ? Pourrait-on faire autrement avec seulement les types que l'on connait pour représenter dans une seule variable le nom de l'animal et son nombre de pattes ?
On pourrait tout aussi bien utiliser un dictionnaire. Le seul avantage de passer par une classe serait une façon un peu plus explicite et compacte de représenter l'information.

4) Des objets « capables »

L'intérêt des objets est qu'en plus de leur associer des données, on peut leur affecter des fonctions capables de traiter ces données. Ce sont les **méthodes** de l'objet.

Une **méthode** est une fonction associée à une classe.



Sans en voir le mécanisme, nous en avons déjà utilisé.
Quand on fait par exemple `tab.append(21)` pour ajouter le nombre `21` au tableau `tab`, on appelle la méthode `append` de l'objet `tab`.

Exemple :
Pour la classe `Animal`, on pourrait définir une méthode « `cri` » qui joue ou affiche le cri de l'animal. Ou une méthode « `caracteristiques` » qui affiche toutes les caractéristiques de l'animal.

```
class Animal:
    def __init__(self, nom, famille, nombre_de_pattes="non défini"):
        self.nom = nom
        self.famille = famille
        self.pattes = nombre_de_pattes

    def caracteristiques(self):
        print("nom :", self.nom, "; famille :", self.famille, "; nombre de pattes :", self.pattes)

m = Animal("Milou", "mammifère", nombre_de_pattes=4)
m.caracteristiques()
```

On remarque que la méthode `caracteristiques` est définie comme une fonction à l'intérieur de la classe `Animal` et que son premier argument est « **`self`** » qui fait référence à l'instance de l'objet pour lequel la méthode est appelé. Cet argument :

- ..... ;
- ..... ;
- ..... .

[1] En réalité `__init__` est une méthode d'instance qui *initialise* l'objet mais la *création* de l'instance se fait en réalité dans une méthode `__new__` que nous ne verrons pas ici. Ceci est spécifique à Python et est implémenté différemment dans d'autres langages.

5) Héritage (*hors programme*)

L'héritage est un concept fondamental en POO. C'est en grande partie ce qui fait la puissance de ce type de programmation.

L'héritage est le procédé par lequel une classe peut reprendre l'ensemble des attributs et des méthodes d'une autre classe.

L'héritage permet de créer des sous-classes d'une classe principale (de « spécialiser » une classe). On dit de la sous-classe que c'est la **classe fille** et la classe dont elle hérite s'appelle **classe mère**, **classe de base** ou encore **super-classe**.

On pourrait ainsi créer une classe `Mammifere` qui hériterait de la classe `Animal` et une classe `Felin` qui hériterait de la classe `Mammifere`.

```
class Mammifere(Animal):
    def __init__(self, nom, nombre_de_pattes="non défini"):
        super().__init__(nom, "mammifère", nombre_de_pattes=nombre_de_pattes)
        self.mamelles = True

m = Mammifere("Milou", nombre_de_pattes=4)
m.caracteristiques()
```

On remarque que :

- L'héritage se fait en précisant le nom de la classe dont on hérite comme argument de la déclaration de classe.
- On a ici redéfini (on dit « **surchargé** ») le constructeur de la classe et on a utilisé le mot-clé `super()` qui permet de faire référence à la classe mère. Ainsi `super().__init__` fait référence au constructeur de la classe mère (celui de la classe `Animal`). Utiliser le constructeur de la classe mère permet d'une part de ne pas avoir à ré-écrire le code existant et surtout de rester compatible avec une ré-écriture de la classe de base.
- On peut faire appel à la méthode `caracteristique` sur un objet de type `Mammifere` car la classe `Mammifere` a hérité de toutes les méthodes de sa classe mère.
- Il est possible de rajouter des attributs et des méthodes à la classe fille. Ici on a rajouté l'attribut `mamelles` et on pourrait rajouter par exemple une méthode `allaite()`.
- Il est également possible de redéfinir une méthode de la classe mère pour mieux l'adapter à la classe fille. Dans ce cas la nouvelle écriture de la méthode utilise souvent la méthode de la classe mère (en utilisant le mot-clé `super()`) en y ajoutant certains points. On parle dans ce cas de **polymorphisme**.

L'héritage permet de factoriser le code au maximum en ré-utilisant le code produit pour la classe mère dans les objets de la classe fille.

III - Premier objet

Nous allons faire un premier programme en POO. Ici, nous souhaitons créer des formes géométriques sur l'écran que nous pourrons par la suite déplacer.

1) Classe `Forme`

On va d'abord écrire le code de la classe `Forme`. Cette classe sera la classe mère pour toutes les formes que nous programmerons ensuite, mais elle ne sera jamais utilisée telle-quelle : elle permet de définir une **interface** commune à toutes les classes.

Voici le squelette de cette classe :

```
class Forme():
    """Classe permettant la représentation de formes sur un écran avec le module
       turtle. La classe en elle-même n'a pas la capacité de tracer quoi que ce
       soit mais sert de classe de base pour des formes concrètes."""
    COULEUR_FOND = "white"
    def __init__(self,x, y, angle, couleur):
        """Crée un objet de type forme. On précise sa position x et y ainsi que
           son angle et sa couleur."""




    def trace(self, couleur=None):
        """Trace la forme à sa position en utilisant sa couleur propre ou une
           autre couleur si celle-ci est précisée"""



    def deplace(self, dx, dy):
        """Déplace la forme d'un décalage dx en abscisse et dy en ordonnée.
           La forme est effacée en la retraçant avec la couleur de fond, puis
           retracée avec sa couleur à la nouvelle position."""




    def tourne(self, d_theta):
        """Tourne la forme sur place d'un angle d_theta.
           La forme est effacée en la retraçant avec la couleur de fond, puis
           retracée avec sa couleur à la nouvelle position."""



    def perimetre(self):
        """Renvoie le périmètre de la forme (en pixels). Le calcul dépendant
           de la nature de la forme, cette fonction est à implémenter dans la
           classe fille"""
        return None
```

Application 1 :
Compléter le squelette fourni en écrivant le code minimal de la classe mère `Forme`.

2) Classe `Polygone`

On va créer une première sous-classe de `Forme` capable de gérer des polygones réguliers. On appellera cette classe `Polygone` et elle devra hériter de `Forme`. En plus des paramètres fournis à `Forme`, le constructeur de `Polygone` prend deux arguments `cote` et `nb_segments`, deux entiers qui désignent respectivement la longueur d'un segment (en pixels) et le nombre de segments du polygone.
Par exemple un carré rouge de 50 pixels de côté situé aux coordonnées x=10, y=20 sera créer avec l'appel :

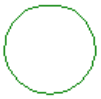
```
carre = Polygone(10, 20, 0, "red", 50, 4)
```

Application 2 :
1) Quelles sont les méthodes de la classe `Forme` que l'on doit surcharger pour écrire la classe `Polygone` ?
2) Ecrire la classe `Polygone` qui hérite de `Forme`.
3) Tester la classe en créant puis en affichant le carré rouge cité en exemple.
4) Utiliser les méthodes `deplace` et `tourne` pour déplacer et faire pivoter le carré.
5) Vérifier que la méthode `perimetre` donne bien le bon résultat sur le carré rouge puis avec d'autres polygones.

3) Classe `Cercle`

Créons maintenant une classe Cercle qui hérite de Forme et permet de manipuler des ronds vides.
Dans le cas d'un cercle, le constructeur prend uniquement en argument x, y, rayon et couleur. Les trois premiers étant des entiers et le troisième une couleur (donc une chaîne ou un tuple).

Par exemple `Cercle(-20, -10, 30, "green")` crée un cercle vert de 30 pixels de rayon dont le centre est en x=-20, y = -10.



Application 3 :
1) Quelles sont les méthodes de la classe `Forme` que l'on doit surcharger pour écrire la classe `Cercle` ?
2) Une des méthodes de `Forme` n'a pas besoin d'être ré-écrite, mais dans le cas d'un cercle, elle peut être (grandement) simplifiée. Quelle est cette méthode ?
3) Ecrire la classe `Cercle` qui hérite de `Forme` (on pourra utiliser la fonction `circle` de turtle).
4) Tester la classe en créant puis en affichant le cercle vert cité en exemple.
5) Utiliser la méthode `deplace` pour déplacer le cercle.
6) Vérifier que la méthode `perimetre` donne bien le bon résultat sur le cercle vert.

IV - Encapsulation

Les différents types de python correspondent à des classes et possèdent donc des attributs et des méthodes.
Prenons l'exemple d'un objet de type tableau (`list`) et examinons ses caractéristiques avec la commande `dir` :

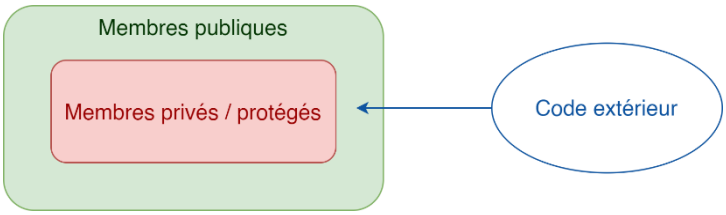
```
>>> t = []
>>> dir(t)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

On remarque que de très nombreux items dans la liste des attributs et méthodes de l'objet `t` sont suivis et précédés de deux sous-tirets « `__` ». Ces items correspondent à des méthodes que l'on n'a jamais utilisées sur les objets de type `list` et pour cause : ce sont des attributs et méthodes **internes** de la classe `list`.

En python, tous les éléments (attributs, méthodes) dont le nom commence par un ou deux sous-tirets sont déclarés [2]être privés : ils ne doivent pas être utilisés en dehors d'appels internes à la classe.

[2] Python ne contient pas de mécanisme de protection des attributs et méthodes privées : c'est juste une convention que les programmeurs sont censés respecter, mais il est toujours possible d'accéder tout de même à ces variables/méthodes depuis l'extérieur de la classe.

Cette normalisation (le ou les sous-tirets) permet **l'encapsulation** :
La classe n'expose à ses utilisateurs que les méthodes publiques qui sont décrites dans son **interface**. Ses attributs et méthodes internes ne sont pas accessibles de l'extérieur.



Ceci garantie un certain niveau de sécurité car les programmeurs externes ne vont pas directement modifier les attributs ou appeler des méthodes spéciales et permet aussi de faire évoluer le code de la classe (en modifiant la façon de faire pour davantage d'efficacité par exemple ou pour s'adapter à de nouveaux standards) sans que les utilisateurs de la classe ne soient affectés (puisque le comportement externe ne change pas).
Plus clairement, il y a 3 statuts possible pour les méthodes et les attributs d'une classe :

- Public : ils commencent par une lettre et ne sont donc pas précédés d'un caractère « _ »
  Ces méthodes sont librement accessibles de l'extérieur et sont décrites dans l'interface de la classe.
- Protégés : ils commencent par un unique signe « _ »
  Ces méthodes et attributs ne sont accessibles que pour les fonctions et classes du module (dans le même fichier .py ou ses fichiers dérivés). Ce statut est peu utilisé en python, mais davantage dans d'autres langages de programmation.
- Privés : ils commencent par deux caractères de soulignement : « __ »
  Les attributs et méthodes privés ne sont accessible que pour les méthodes de la classe à laquelle ils appartiennent.

Si on souhaite que l'utilisateur puisse modifier certains attributs de la classe ou voir leur valeur, on écrit des méthodes publiques particulières :

Les ..... (ou ..... en français) qui renvoient la valeur d'un attribut (généralement nommés `get_nomAttribut`. Ex : `get_x()`).

Les ..... (ou ..... en français) qui permettent de fixer la valeur d'un attribut (généralement nommés `set_nomAttribut`. Ex : `set_angle(valeur)`).

Il existe des façons plus « pythonique » de gérer des attributs publics tout en maintenant l'encapsulation, mais cela dépasse la portée de ce cours.

V - Les attributs et les méthodes spéciales

On a vu plus haut que les méthodes et attributs privés doivent *commencer* par « __ », mais on a également vu dans l'exemple de la classe List que ces méthodes et attributs *finissaient* également par « __ ».

Ces méthodes et attributs ont une signification particulière pour python et sont présent dans de nombreuses classes

(ce que l'on peut vérifier avec `dir(dict)` ou `dir(str)` par exemple). On les appelle .....

ou encore ..... .

Elles permettent d'utiliser des commandes python comme `len()`, `+`, `>` ou `abs()` directement avec les instances d'une classe qu'on a programmée.
Voyons deux méthodes spéciales :

1) `__repr__`

Cette méthode est appelée à chaque fois qu'on cherche à représenter l'instance d'une classe. Par exemple lorsque le débogueur affiche le contenu d'une variable ou qu'on utilise la commande `print` (voir dans les références la différence entre `__repr__` et `__str__`).
Il est donc toujours intéressant d'implémenter cette méthode dans nos classes car cela aidera au débogage.

2) `__lt__`

Cette fonction correspond à l'opérateur `<` de python (lt signifie **L**ess **T**han). Lorsqu'on écrit `a < b` en réalité python appelle la fonction `__lt__` de l'objet `a` en lui fournissant en argument l'objet `b`.
Si on implémente cette fonction, il est alors possible d'utiliser un algorithme de tri sur les instances de nos objets.

3) Exemple

Voici un exemple avec une classe Couleur qui permet le codage des couleurs avec leur nom, composantes rvb et longueur d'onde.

```
class Couleur():
  DEF_COULEURS={'BLEU':((0,0,255),470), 'ROUGE':((255,0,0),650),
                'VERT':((0,255,0),540), 'CYAN':((0,255,255),505),
                'JAUNE':((255,255,0),580),'ORANGE':((255,150,0),600),
                'VIOLET':((100,10,200),415)}
  def __init__(self, texte_couleur:str):
    texte_couleur = texte_couleur.upper() # Pour indifférencier la case
    self._texte = texte_couleur
    if texte_couleur in Couleur.DEF_COULEURS.keys():
      self._rvb = Couleur.DEF_COULEURS[texte_couleur][0]
      self._lambda = Couleur.DEF_COULEURS[texte_couleur][1]
    else:
        raise Exception("Couleur inconnue : " + texte_couleur)
```

```
  def __repr__(self):
    return self._texte + " : RVB = " + str(self._rvb) \
           + " λ = " + str(self._lambda)

  def __lt__(self, other):
    if type(other) != Couleur:
      raise TypeError("Impossible de comparer une couleur avec le type"\
                      + str(type(other)))
    else:
      return self._lambda < other._lambda

t = [Couleur('bleu'), Couleur('jaune'), Couleur('Vert'),
     Couleur('Violet'), Couleur('rouge'), Couleur('orange')]
print("Avant le tri :", t)
print("Après le tri :", sorted(t))
```

On voit que grâce à l'implémentation de `__repr__`, on peut afficher simplement les informations d'une couleur et qu'avec la programmation de `__lt__`, on peut utiliser l'instruction `sorted` pour trier une liste de couleur.

VI - Compléments

1) Attributs et méthodes de classe

Les attributs et méthodes que nous avons décrits jusqu'ici sont en réalité des attributs et des méthodes **d'instance** : ils sont associés à une instance de la classe (rappel : les attributs et les méthodes sont les mêmes pour toutes les instances de la classe, mais les *valeurs* des attributs sont propres à chaque instance).
Il existe aussi des attributs et des méthodes **de classe** qui sont attachés à la classe elle-même et non à ses instances. Cela peut être utile pour gérer des choses qui sont globales pour tous les objets de la classe comme par exemple le nombre d'objets de la classe instanciés.
On y accède comme aux attributs et méthodes d'instance, mais ..... .

Exemple :

```
class Ennemi:
    _nb_ennemis = 0   # Garde trace du nombre d'instances de la classe

    def __init__(self, nom:str):
        self.nom = nom
        Ennemi._nb_ennemis += 1   # Incrémente le compteur d'instances

    def version():
        return "Classe Ennemi - v1.02"

    def effectif():
        return Ennemi._nb_ennemis

e1 = Ennemi("Sauron")
e2 = Ennemi("Morgoth")
print(Ennemi.effectif())   # Affiche 2 qui est le nombre d'ennemis crées
print(Ennemi.version())
```

On remarque que les méthodes de classe n'ont pas l'argument `self` comme premier argument (ce qui est logique puisqu'elles ne dépendent pas d'une instance mais de la classe elle-même).

Un ..... est donc un attribut lié à la classe. Il existe à partir du moment où la classe est définie, même s'il n'existe aucune instance de cette classe.

Une ..... est une fonction liée à la classe. De même elle n'est pas liée à une instance et n'utilise donc pas le mot-clé `self`.

Lorsqu'on utilise le mot « attribut » ou « méthode », sans précision, il s'agira toujours d'attribut ou de méthode **d'instance**.

2) Interface

La notion d'interface est importante en POO. Dans certains langages, il est d'ailleurs possible de spécifier des interfaces ou classes abstraites qui ne contiennent aucun code, mais spécifient les noms des attributs d'une classe ainsi que les signatures de ses méthodes.
La connaissance de l'interface d'une classe est suffisante pour pouvoir l'utiliser et manipuler ses instances.
Grâce à l'encapsulation, on pourra changer l'implémentation (la façon dont le code de la classe est écrit) de la classe facilement. Tant qu'on garde la même interface, la nouvelle version restera compatible avec les programmes qui l'utilisent ou la sous-classent.
On essaie généralement de ne mettre qu'un minimum d'attribut dans l'interface (donc publics) et on garde la majorité (ou la totalité) des attributs privés. De même pour respecter le principe de l'encapsulation, l'interface ne contiendra que les méthodes publiques : les méthodes privées ou protégées ne font pas partie de l'interface.

3) Attributs et arguments de la fonction `__init__`

On considère le programme suivant :

```
class Pompier:
    def __init__(self, nom:str, taille:int, poids:int, equipement:list):
        self.nom = nom
        self.taille = taille
        self.poids = poids
        self.equipement = equipement

    def ajoute_equipement(self, ajout):
        if type(ajout) == list:
            self.equipement += ajout
        elif type(ajout) == str:
            self.equipement.append(ajout)
        else:
            raise(TypeError)

    def __repr__(self):
        return self.nom + " : " + str(self.taille) + "cm, " + str(self.poids) +\
               "kg, équipement : " + str(self.equipement)
equipement_de_base = ["Uniforme", "Casque"]
sam = Pompier("Sam", 180, 75, equipement de base)
annie = Pompier("Annie", 174, 62, equipement_de_base)
sam.ajoute_equipement("Lance à incendie")
annie.ajoute_equipement("Échelle")
print(sam)
print(annie)
```

Application 4 :
1) Que fait la méthode `ajoute_equipement` (soyez complet dans la réponse) ?
2) Quel est le résultat attendu en principe ? Sans exécuter le programme, dire ce qu'il va afficher à la fin.
3) Copier-coller le programme et exécutez-le. Expliquez le résultat observé.
4) Conclure : à quoi faut-il faire attention lors du passage de paramètres à la fonction d'initialisation des instances ?

Références :
Différence entre objet et instance (en anglais) : https://stackoverflow.com/questions/3323330/difference-between-object-and-instance/42753129#42753129
Encapsulation et attributs : https://realpython.com/python-getter-setter/
Méthodes spéciales : https://openclassrooms.com/fr/courses/4425111-perfectionnez-vous-en-python/4463810-decouvrez-les-methodes-speciales et https://pythonforge.com/les-methodes-speciales-pour-renforcer-vos-classes-python/
`__repr__` et `__str__` : https://www.python-engineer.com/blog/difference-between-str-and-repr/

Exercices – **Programmation objet**

Exercice 1 : D&D

Pour un jeu de type aventure médiéval fantastique, on souhaite créer des classes pour représenter les personnages. On prévoit une classe `Personnage` dont les instances auront un nom (« `nom` »), un nombre de points de vie (« `pv` »), un score d'attaque (« `att` ») et une classe d'armure (« `ca` ») définis à la création.

1) Ecrire le constructeur de la classe qui doit permettre d'initialiser ses attributs (qui seront fourni en argument de la fonction) ainsi qu'une méthode `caracteristiques` qui affiche les attributs du personnage. Instancier quelques personnages pour tester la classe.

2) Ecrire une méthode `attaque(adversaire)` qui prend en argument (en plus de `self`…) un autre objet `Personnage`. Cette méthode ajoute au score d'attaque du personnage un nombre au hasard entre 0 et 10. Si cette somme dépasse la classe d'armure de l'adversaire, il perd 1 à 6 points de vie. La méthode renvoie `True` si l'attaque a réussie (et infligé des dégâts à l'adversaire) et `False` sinon. Vous devez écrire cette méthode en manipulant directement les attributs et sans écrire d'autres méthodes.

On voudrait prendre en compte le fait que les personnages peuvent mourir si le nombre de points de vie (`pv`) devient inférieur ou égal à zéro. Un personnage mort ne doit plus pouvoir agir, mais conserve ses attributs et son instance continu d'exister (dans un jeu vidéo médiéval fantastique il sera peut-être possible de le ressusciter plus tard…).

3) Quel(s) problème(s) rencontre-t-on pour réaliser cette évolution ?

Pour solutionner cela, on ajoute un attribut booléen `est_vivant`, initialisé à `True` aux instances de la classe `Personnage`. On ajoute également des accesseurs et mutateurs à l'attribut `pv`.

4) Modifier la classe `Personnage` en implémentant ces changements et mettre en place les évolutions prévues plus haut (possibilité de mourir pour un personnage).

En réalité cette solution (accesseurs et mutateurs) n'est pas vraiment satisfaisante car si on souhaite par exemple implémenter un sort d'invulnérabilité qui empêche toute baisse de points de vie à un personnage pendant un laps de temps, cela ne sera pas possible avec le système actuel.

5) Quelle solution proposez-vous pour solutionner ce problème ?

6) Ecrire une classe `Gueriseur` qui hérite de `Personnage`. Ajouter un attribut « `energie` » (points d'énergie) aux membres de cette classe qui sera initialisé à 50 pour toutes les instances de la classe (mais qui pourra varier pour chacune).

7) Ajouter à la classe `Guerisseur` une méthode `guerir(personnage)` qui prend en argument un autre objet de la classe `Personnage` et lui augmente ses points de vie d'un nombre aléatoire entre 1 et 10. Le guérisseur perd alors le même nombre de points d'énergie. Si les points d'énergie deviennent négatifs, le personnage n'est plus capable de guérir et sa méthode `guerir` n'a plus d'effet.

8) Écrire une classe `Mage` qui hérite de `Personnage`. Ajouter un attribut « `mana` » (points de magie, initialisés à 50) et « `magie` » (score de magie qui devra être précisé dans les paramètres de création de l'objet juste après nom, pv, …).

Les personnages de la classe `Mage` attaquent différemment des autres : c'est leur score de magie qui doit être ajouté à la place de leur score d'attaque et ils enlèvent 2 à 10 points de vie à leur adversaire. Ces points de dégâts sont soustraits aux points de magie (`mana`). Si les point de magie deviennent négatif il n'est plus possible au mage d'attaquer avec sa magie et il réalise alors une attaque classique (comme les autres instances de la classe `Personnage`).

9) Expliquer comment implémenter cette contrainte dans la classe `Mage` puis écrire le code nécessaire.

10) Sans exécuter le programme, dire s'il est possible pour un guérisseur de guérir un autre guérisseur ? Un mage ? Vérifier en créant quelques instances.

Exercice 2 : Angles

Définir une classe `Angle` pour représenter un angle **en degrés**. Cette classe contient un unique attribut, `angle`, qui est un entier. On demande que, quoi qu'il arrive, l'égalité $0° \leq$ `angle` $\leq 360°$[3] reste vérifiée.

1) Ecrire le constructeur de cette classe.

2) Ajouter une méthode `__repr__` qui renvoie une chaîne de caractère de la forme `"60°"` (en ajoutant donc bien le signe « `°` » après la valeur de l'angle). Observer son effet en construisant un objet de la classe `Angle` puis en l'affichant avec `print`.

3) Ajouter une méthode `ajoute` qui reçoit un autre angle en argument (un objet de la classe `Angle`) et l'ajoute au champ `angle` de l'objet. Attention à ce que la valeur de l'angle reste bien dans l'intervalle autorisé.

4) Ajouter deux méthodes `cos` et `sin` pour calculer respectivement le cosinus et le sinus de l'angle. On utilisera pour cela les fonctions de la bibliothèque math. Attention : il faut convertir l'angle en radians (en multipliant par $\pi/180$) avant d'appeler les fonctions trigonométriques de math.

5) Implémenter une fonction `__add__` décrite ci-contre, puis essayer de définir deux angles `a1` et `a2` et d'afficher `a1 + a2`. Qu'observe-t-on ?

```
def __add__(self, other):
    if type(other) == Angle:
        return Angle(self.__angle + other.__angle)
    elif type(other) == int or type(other) == float:
        return Angle(self.__angle + other)
    else:
        raise TypeError
```

Exercice 3 : Fractions

Définir une classe `Fraction` pour représenter un nombre rationnel. Cette classe possède deux attributs, `num` et `denom`, qui sont des entiers et désignent respectivement le numérateur et le dénominateur de la fraction représentant le nombre. On demande plus particulièrement que le dénominateur soit un entier strictement positif.

1) Ecrire le constructeur de cette classe. Le constructeur doit lever une `ValueError` si le dénominateur fournit n'est pas strictement positif.

2) Ajouter une méthode `__repr__` qui renvoie une chaîne de caractère de la forme `"12/35"`, ou simplement `"12"` lorsque le dénominateur vaut 1.

3) Ajouter des méthodes `__eq__` (test d'égalité) et `__lt__` (test d'infériorité) qui reçoivent une deuxième fraction en argument et renvoie `True` si la première fraction représente respectivement un nombre égal ou un nombre strictement inférieur à la deuxième fraction (ou `False` dans le cas contraire).

4) Ajouter des méthodes `__add__` et `__mul__` qui reçoivent une deuxième fraction en argument et renvoient une nouvelle fraction représentant respectivement la somme et le produit des deux fractions.

5) Effectuer les tests ci-contre. Qu'auraient donné ces tests si on avait utilisé les flottants plutôt que la classe `Fraction` ?

```
assert Fraction(1,3) == Fraction(5, 3) + Fraction (-4,3)
assert Fraction(15,77) == Fraction(3,7) * Fraction(5,11)
```

Bonus : S'assurer que les fractions sont toujours sous forme réduite (il faut calculer le pgcd de `num` et `denom`).

[3] On pourra utiliser l'opérateur « `%` » pour obtenir l'angle « modulo » 360.

Exercice 4 : Intervalles

Définir une classe `Intervalle` représentant des intervalles de nombres. Cette classe possède deux attributs `inf` et `sup` représentant respectivement l'extrémité inférieure et supérieure de l'intervalle. Les deux extrémités sont considérées comme incluses dans l'intervalle. Tout intervalle avec `sup` < `inf` représente l'intervalle vide.

1) Ecrire le constructeur de cette classe et une méthode `estvide` renvoyant `True` si l'objet représente un intervalle vide et `False` sinon.

2) Ajouter des méthodes `__len__` renvoyant la longueur de l'intervalle (l'intervalle vide a une longueur nulle) et `__contains__` testant l'appartenance d'une valeur `v` à l'intervalle.

3) Ajouter une méthode `__eq__` permettant de tester l'égalité de deux intervalles avec `==` et une méthode `__le__` permettant de tester l'inclusion d'un intervalle dans un autre avec `<=`. Attention : toutes les représentations de l'intervalle vide doivent être considérées égales, et incluses dans tout intervalle.

4) Ajouter des méthodes `intersection` et `union` calculant respectivement l'intersection de deux intervalles et le plus petit intervalle contenant l'union de deux intervalles (l'intersections est bien toujours un intervalle, alors que l'union ne l'est pas forcément). Ces deux fonctions doivent renvoyer un nouvel intervalle sans modifier leurs paramètres.

Par exemple `Intervalle.intersection(Intervalle(3,7), Intervalle(4,9))` donnera [4, 7]. `Intervalle.union(Intervalle(3,7), Intervalle(4,9))` donnera [3, 9] et `Intervalle.union(Intervalle(-2,3), Intervalle(4,9))` donnera [-2, 9]

TNSI	Représentation des données Programmation objet	Vocabulaire – **Programmation objet**

Définir les mots suivants :

Accesseur :
Attribut :
Classe :
Encapsulation :
Héritage :
Instance :

Interface :
Méthode :
Méthode magique :
Mutateur :
Objet :
Polymorphisme :