



Python

Programmation Orientée Objet

Par Richard BONNAMY



Sommaire

Sommaire

- Introduction
- Les classes
- Appels de méthode
- Méthodes de classe
- Méthodes static
- Encapsulation
- Getters et setters
- Héritage
- Héritage – compléments
- Classes et méthodes abstraites
- Les tuples
- Les dictionnaires
- Les sets
- Les itérateurs
- `__str__` et `__repr__`
- Égalité de 2 instances
- Méthode `__hash__` pour les sets
- Trier vos listes
- Gérer les exceptions
- Documentation des classes



Introduction

Python- Qu'est-ce que la POO ?

- La **programmation orientée objet** (ou **POO** en abrégé) correspond à une autre manière d'imaginer, de construire et d'organiser son code.
- La **programmation orientée objet** repose sur le concept d'objets qui possèdent des variables et fonctions qui leur sont propres.
- Les objectifs principaux de la **programmation orientée objet** sont de nous permettre de créer des scripts plus clairs, mieux structurés, plus modulables et plus faciles à maintenir et à déboguer.

Python- Un langage orienté objet

- **Python** est un langage résolument **orienté objet**, ce qui signifie que le langage tout entier est construit autour de la notion d'objets.
- En **Python** tout est objet: les types **str**, **int**, **list**, etc. sont avant tout des objets, les **fonctions** sont des **objets**, etc.
- Il est indispensable de comprendre cette dimension orientée objet.

Python- Qu'est-ce qu'un objet ?

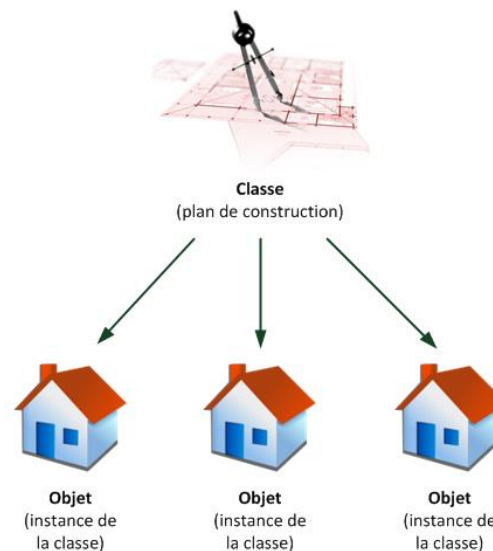
- Dans la vie réelle, un **objet** possède des **caractéristiques** (porteuses d'une valeur) et nous permet de **réaliser des actions**.
 - Par exemple, une voiture possède des caractéristiques : une marque, un modèle, une longueur, une largeur, un poids, etc.
 - Une voiture peut également réaliser des actions comme : démarrer, avancer, reculer, freiner, changer de direction
- Le concept d'objet en informatique s'inspire de la vie réelle.
- On appelle **objet** un bloc de code qui **possède ses propres variables** (qui sont l'équivalent des caractéristiques des objets de tous les jours) et **fonctions** (qui sont les actions réalisables avec l'objet).



Les Classes

Python- Les Classes

- En POO, on distingue la notion de **classe** de celle d'**objet**.
- Bien qu'intimement liées, ces notions ne représentent pas tout à fait la même chose.
- Une classe représente la définition de la notion, le plan à partir duquel les objets sont créés.
- Vous pouvez vous représenter la **classe** comme le **plan d'une maison** réalisé par un architecte et les **objets** comme les **maisons créées à partir du plan**.



Python- Création d'une première classe

- Pour créer une nouvelle classe en Python on utilise le mot clef **class** suivi du nom de la classe.
- Ci-dessous nous créons la classe Utilisateur :

```
class Utilisateur:  
  
    def __init__(self):  
        print("Un nouvel utilisateur vient de naître")
```

- Notez que par convention **le nom d'une classe commence toujours par une majuscule.**

Python- Création de premiers objets

- A noter que cette classe Utilisateur possède "une fonction" `__init__()` **un peu étrange**. C'est une fonction d'initialisation. Nous y reviendrons.

```
class Utilisateur:  
  
    def __init__(self):  
        print("Un nouvel utilisateur vient de naître")
```

- Les fonctions situées dans les classes sont appelées **méthodes**.

Python- Création de premiers objets

- Intéressons-nous pour l'instant à la création d'objets de **type Utilisateur**.

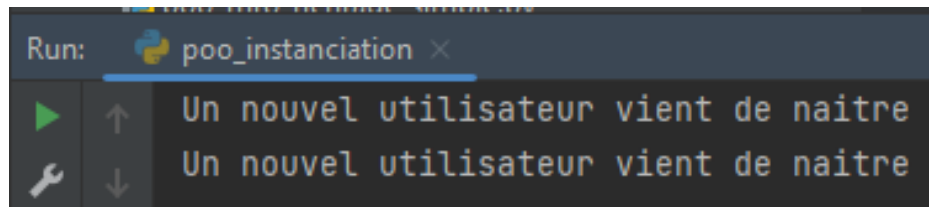
```
ut1 = Utilisateur()  
ut2 = Utilisateur()
```

- La syntaxe est d'invoquer le nom de la classe suivi de parenthèses, pour créer un objet de type Utilisateur.
- Ici à partir de la classe Utilisateur nous avons créé 2 objets : ut1 et ut2

Python- Exécution du code

- Que se passe t'il lorsque j'exécute le code précédent ?

```
class Utilisateur:  
  
    def __init__(self):  
        print("Un nouvel utilisateur vient de naître")  
  
ut1 = Utilisateur()  
ut2 = Utilisateur()
```



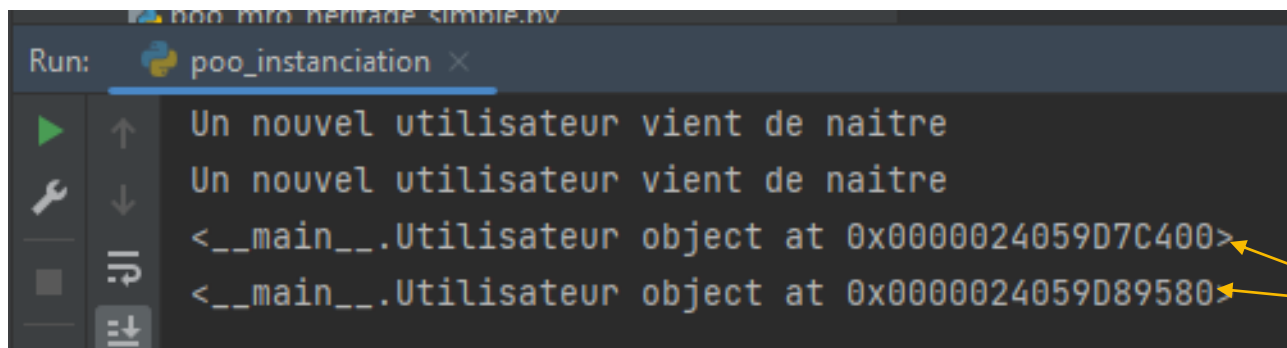
The screenshot shows a terminal window titled "Run: poo_instanciation". It contains two lines of output, each preceded by a green play button icon and an upward arrow, indicating the execution of the code. The output is "Un nouvel utilisateur vient de naître" repeated twice.

- Le code situé dans la méthode `__init__` a été exécuté.
- Cette **méthode** est en effet **la méthode d'initialisation** de la classe appelée à **chaque nouvelle création d'utilisateur**.

Python- Affichage des objets

- Que se passe t'il lorsque j'affiche ut1 et ut2 ?

```
class Utilisateur:  
  
    def __init__(self):  
        print("Un nouvel utilisateur vient de naitre")  
  
ut1 = Utilisateur()  
ut2 = Utilisateur()  
  
print(ut1)  
print(ut2)
```



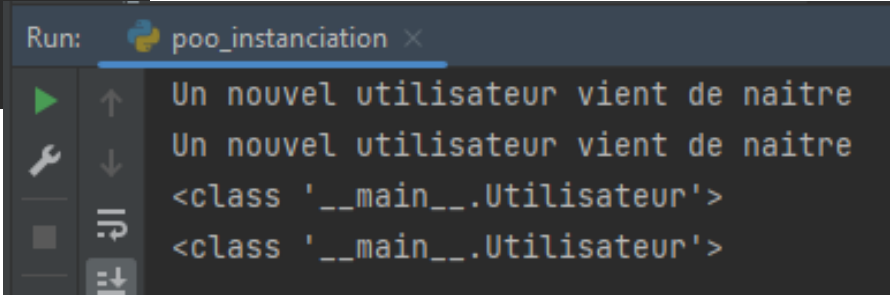
```
Run: poo_instanciation ×  
Un nouvel utilisateur vient de naitre  
Un nouvel utilisateur vient de naitre  
<__main__.Utilisateur object at 0x0000024059D7C400>  
<__main__.Utilisateur object at 0x0000024059D89580>
```

Adresses mémoires

Python- Affichage des types des objets

- Que se passe t'il lorsque j'affiche le type de ces objets ?

```
class Utilisateur:  
  
    def __init__(self):  
        print("Un nouvel utilisateur vient de naitre")  
  
ut1 = Utilisateur()  
ut2 = Utilisateur()  
  
print(type(ut1))  
print(type(ut2))
```



```
Run: poo_instanciation x  
Un nouvel utilisateur vient de naitre  
Un nouvel utilisateur vient de naitre  
<class '__main__.Utilisateur'>  
<class '__main__.Utilisateur'>
```

- Le **type** de ces objets est **Utilisateur**.
- **__main__** est le **scope d'exécution** (contexte) du script.

Python- Instance

- Lorsqu'on crée un objet à partir d'une classe comme ceci, on dit également qu'on **instancie** une classe.
- Ici, on instancie deux fois notre classe et on place le résultat dans deux variables **ut1** et **ut2**.
- **ut1** et **ut2** sont aussi appelées **instances de la classe Utilisateur**.

Python- Les Classes

- Nos objets vont pouvoir posséder des attributs qui leur sont propres :

```
class Utilisateur:

    def __init__(self):
        print("Un nouvel utilisateur vient de naître")

ut1 = Utilisateur()
ut2 = Utilisateur()

ut1.nom = "BEZOS"
ut1.age = 59

print(ut1.nom)
print(ut2.nom )
```

```
Traceback (most recent call last):
  File "C:\Users\RichardBONNAMY\PycharmProjects\preparation
    print(ut2.nom )
AttributeError: 'Utilisateur' object has no attribute 'nom'
```

- Dans cet exemple, on a ajouté **un nom et un age** à ut1 mais **pas à ut2**
- A l'exécution, l'interpréteur nous dit une chose intéressante: **la classe Utilisateur n'a pas d'attribut nom.**
- Cette manière de définir les attributs d'une classe n'est pas la bonne manière de faire. Nous y reviendrons dans la suite.

Python- Attributs et méthodes d'instances

- Pour désigner les variables et les fonctions dont les classes disposent, Python utilise les termes **attributs d'instance** et **méthodes d'instance**

```
class Utilisateur:

    def __init__(self):
        print("Un nouvel utilisateur vient de naître")

ut1 = Utilisateur()
ut2 = Utilisateur()

ut1.nom = "BEZOS"
ut1.age = 59

print(ut1.nom)
print(ut2.nom )
```

- La classe Utilisateur possède la méthode d'instance `__init__`

Python- Initialisation des objets

- Précédemment nous avons défini la classe Utilisateur et avons créé deux objets **ut1** et **ut2** à partir de cette classe :

```
class Utilisateur:

    def __init__(self):
        print("Un nouvel utilisateur vient de naître")

ut1 = Utilisateur()
ut2 = Utilisateur()
```

- **Utilisateur()** s'appelle le **constructeur** de la classe Utilisateur
- Le **constructeur invoque** la méthode **__init__** qui participe à la création de l'objet (mais ce n'est pas la seule).

Python- `__init__()` avec des paramètres

- Il est possible de passer **des paramètres à la méthode `__init__`**
- Dans l'exemple ci-dessous la méthode `__init__` reçoit des paramètres (à adapter selon les besoins) :
 - **nom** et **age** dans l'exemple ci-dessous.
 - **self** est un paramètre technique, implicite, dont on va parler dans la diapo suivante.

```
class Utilisateur:  
  
    def __init__(self, nom, age):  
        self.nom = nom  
        self.age = age
```

- **nom** et **age** sont appelés **attributs d'instance**

Python- Explications sur `__init__()`

- Comme on l'a vu précédemment, la méthode `__init__()` accepte **trois paramètres** en entrée qu'on a ici nommé **self**, **nom** et **age**.

```
def __init__(self, nom, age):  
    self.nom = nom  
    self.age = age
```

- **self** est un paramètre qui représente l'objet cible, i.e. l'objet en cours de construction
- **self** est implicite car on doit l'ignorer lors de l'instanciation (cf. exemple ci-dessous).
- **Exemple : self** représente **jeff_bezos** dans le 1^{er} cas et **elon_musk** dans le 2nd cas :

```
jeff_bezos = Utilisateur("Bezos", 59)  
elon_musk = Utilisateur("Musk", 51)
```

Python- Impact sur le constructeur

- Maintenant que la méthode `__init__` possède 2 paramètres (en plus de `self`), le **constructeur a 2 paramètres obligatoires à renseigner.**

```
def __init__(self, nom, age):  
    self.nom = nom  
    self.age = age
```

- **Exemple :**

```
jeff_bezos = Utilisateur("Bezos", 59)  
elon_musk = Utilisateur("Musk", 51)
```

Python- Que fait exactement le constructeur ?

```
class Utilisateur:
    # constructeur
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

jeff_bezos = Utilisateur("Bezos", 59)
elon_musk = Utilisateur("Musk", 51)
```

- Lorsqu'on invoque le constructeur `Utilisateur(...)` :
- ce dernier commence par invoquer une méthode appelée `__new__()` qui alloue un espace mémoire au nouvel objet,
 - Ensuite il appelle la méthode `__init__()`
 - **En réalité l'interpréteur Python fait ceci :**

```
jeff_bezos = object.__new__(Utilisateur)
Utilisateur.__init__(jeff_bezos, "Bezos", 59)
```

Python- Utilisation des attributs d'instance

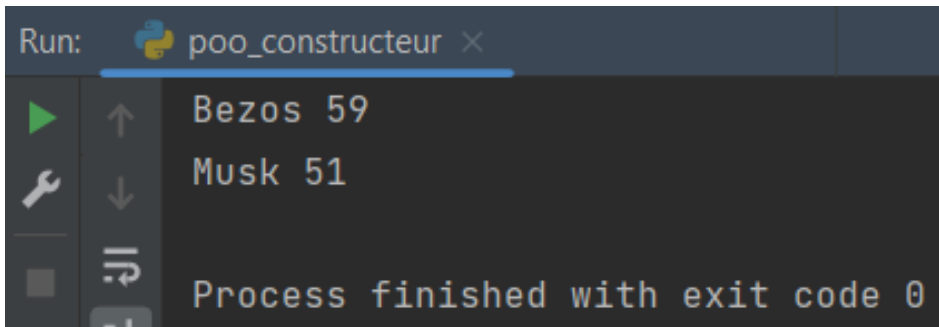
- On peut lire et même modifier les attributs **nom** et **age** de chaque instance :

```
class Utilisateur:
    # constructeur
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
```

```
jeff_bezos = Utilisateur("Bezo", 59)
elon_musk = Utilisateur("Musk", 51)

jeff_bezos.nom = "Bezos"

print(jeff_bezos.nom, jeff_bezos.age)
print(elon_musk.nom, elon_musk.age)
```



```
Run: poo_constructeur x
Bezos 59
Musk 51
Process finished with exit code 0
```


Python- Les attributs de classe

- Il est possible de définir ce qu'on appelle un **attribut de classe**
- A la différence d'un attribut d'instance, un attribut de classe a une **valeur unique indépendante** de chaque objet.

```
class Utilisateur:

    societe = "DIGINAMIC"

    # constructeur
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

jeff_bezos = Utilisateur("Bezo", 59)
elon_musk = Utilisateur("Musk", 51)

print(Utilisateur.societe)
```

- Notez que **cet attribut dépend de la classe** et non de chaque instance.

Python- Modification d'un attribut de classe

- Si j'écris `jeff_bezos.societe = "AMAZON"`
- Est-ce que je modifie `Utilisateur.societe` ?

```
class Utilisateur:

    societe = "DIGINAMIC"

    # constructeur
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

jeff_bezos = Utilisateur("Bezo", 59)
jeff_bezos.societe = "AMAZON"
```

- Réponse ?

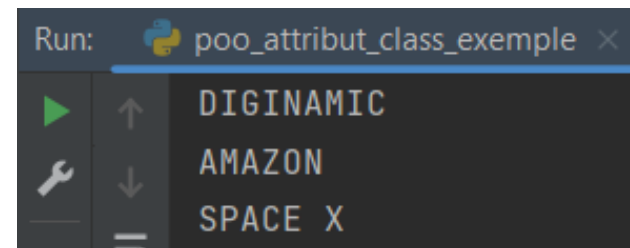
Python- Les attributs de classe

- `jeff_bezos.societe = "AMAZON"` a pour effet de créer un **attribut d'instance** `societe` **différent** de l'attribut de classe `Utilisateur.societe`.

```
jeff_bezos = Utilisateur("Bezo", 59)
elon_musk = Utilisateur("Musk", 51)

jeff_bezos.societe = "AMAZON"
elon_musk.societe = "SPACE X"

print(Utilisateur.societe)
print(jeff_bezos.societe)
print(elon_musk.societe)
```



- A noter que la variable `elon_musk` ne disposera pas de cet attribut d'instance `societe`.
- Pour modifier l'attribut de la classe : `Utilisateur.societe = "AMAZON"`

TP n°1: Classes et instances



Appels de méthode

Python- Les méthodes d'instance

- Il est possible de définir des **méthodes d'instance** avec :
- Le mot clé **def**
 - Le nom de la méthode (ici set_nom)
 - Une liste de paramètres entre parenthèses : **self** est obligatoire en 1^{ère} position.

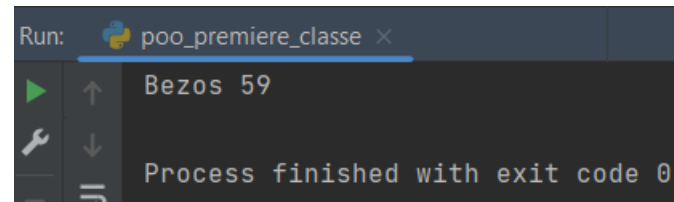
```
class Utilisateur:
    societe = "DIGINAMIC"

    # constructeur
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def set_nom(self, nom):
        self.nom = nom

ut1 = Utilisateur("Gates", 59)
ut1.set_nom("Bezos")

print(ut1.nom, ut1.age)
```



```
Run: poo_premiere_classe x
Bezos 59
Process finished with exit code 0
```

Python- return

- Une méthode peut retourner un résultat, par exemple un résultat de calcul :
 - Dans ce cas la méthode doit utiliser le mot clé **return**.
- Exemple :

```
class Utilisateur:  
  
    def __init__(self, nom, age):  
        self.nom = nom  
        self.age = age  
  
    def dire_son_nom(self):  
        return "Je m'appelle "+self.nom
```

```
ut1 = Utilisateur("Bezos", 59)  
  
identite = ut1.dire_son_nom()  
print(identite)
```

```
Run: poo_methode_instance_return x  
▶ ↑ Je m'appelle Bezos
```

Python- plusieurs return

- Une méthode peut retourner plusieurs résultats, s'il y a plusieurs branches (if elif else)
 - Dans ce cas la méthode peut utiliser un **return** par branche
- Exemple :

```
class Utilisateur:

    def __init__(self, nom, age, societe=None):
        self.nom = nom
        self.age = age
        self.societe = societe

    def dire_son_nom(self):
        if self.societe:
            return "Je m'appelle "+self.nom \
                +" et je travaille chez "+self.societe
        else:
            return "Je m'appelle "+self.nom
```

```
ut1 = Utilisateur("Bezos", 59)
ut2 = Utilisateur("Musk", 50, "Tesla")

print(ut1.dire_son_nom())
print(ut2.dire_son_nom())
```

```
Run: poo_methode_instance_return x
Je m'appelle Bezos
Je m'appelle Musk et je travaille chez Tesla
```


Python- les paramètres facultatifs

- Une méthode peut posséder des paramètres facultatifs à condition de leur fournir une valeur par défaut dans la signature de la méthode
- Exemple :

```
class Utilisateur:  
  
    def __init__(self, nom, age, societe=None):  
        self.nom = nom  
        self.age = age  
        self.societe = societe
```

None est le "zéro" des objets et désigne une absence de valeur.

Pour un nombre on mettra plutôt 0 ou 0.0 et pour un booléen False.

```
ut1 = Utilisateur("Bezos", 59)  
ut2 = Utilisateur("Musk", 50, "Space X")  
print(ut1.societe)  
print(ut2.societe)
```

Pour ut1 on ne valorise pas societe mais on le fait pour ut2

```
Run: poo_param_facultatif x  
None  
Space X
```

Python- les paramètres facultatifs

- On peut avoir plusieurs paramètres facultatifs.
- **Problématique** : comment ne renseigner que les paramètres qui m'intéressent en fonction de mes besoins ?
- **Exemple** :

```
class Utilisateur:  
  
    def __init__(self, nom, age, societe=None, fortune=0)  
        self.nom = nom  
        self.age = age  
        self.societe = societe  
        self.fortune = fortune
```

- Je veux renseigner la fortune pour mon 1^{er} utilisateur mais pas la société
 - Je veux faire l'inverse pour mon 2nd utilisateur.
- Réponse ?

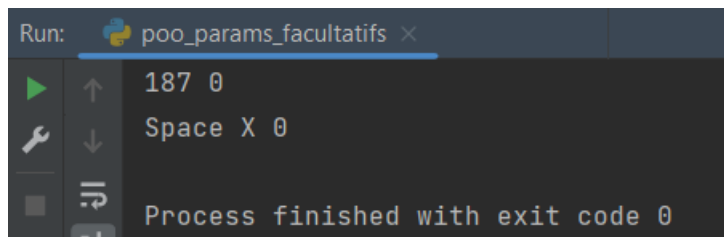
Python- les paramètres facultatifs – Solution 1

➤ Solution 1 : ne marche pas

```
class Utilisateur:

    def __init__(self, nom, age, societe=None, fortune=0):
        self.nom = nom
        self.age = age
        self.societe = societe
        self.fortune = fortune

ut1 = Utilisateur("Bezos", 59, 187)
ut2 = Utilisateur("Musk", 50, "Space X")
print(ut1.societe, ut1.fortune)
print(ut2.societe, ut2.fortune)
```



```
Run: poo_params_facultatifs ×
187 0
Space X 0
Process finished with exit code 0
```

➤ On voit à l'exécution que 187 a valorisé societe au lieu de valoriser fortune

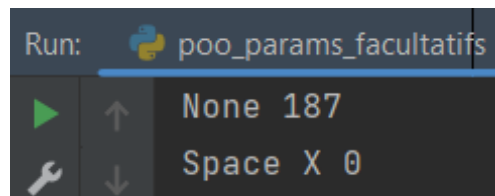
Python- les paramètres facultatifs – solution 2

- Solution 2 : fonctionne mais on renseigne tout

```
class Utilisateur:

    def __init__(self, nom, age, societe=None, fortune=0):
        self.nom = nom
        self.age = age
        self.societe = societe
        self.fortune = fortune

ut1 = Utilisateur("Bezos", 59, None, 187)
ut2 = Utilisateur("Musk", 50, "Space X", 0)
print(ut1.societe, ut1.fortune)
print(ut2.societe, ut2.fortune)
```



Run: poo_params_facultatifs

None 187

Space X 0

- On n'a pas tout à fait répondu à la problématique puisqu'on a valorisé les 4 paramètres et non uniquement ceux dont on a besoin

Python- les paramètres facultatifs – Solution 3

➤ Solution 3 :

```
ut1 = Utilisateur("Bezos", 59, fortune=187)  
ut2 = Utilisateur("Musk", 50, societe="Tesla")
```

- La solution 3, qui répond à notre problématique, consiste à réaliser une affectation dans l'appel du constructeur.
- Dans le premier cas on donne une valeur à fortune et dans le second on donne une valeur à societe.
- A noter que cela fonctionne avec n'importe quelle méthode et fonction.

Python- les hints

- Afin de rendre les méthodes plus faciles à utiliser, il est possible d'indiquer le type attendu pour chaque paramètre.
- On appelle ça des **hints**. Cela n'a aucun caractère contraignant, c'est juste une indication.

```
class Utilisateur:  
  
    def __init__(self, nom: str, age: int):  
        self.nom = nom  
        self.age = age
```

- A la droite de chaque paramètre on ajoute ":" suivi du type attendu.

Python- hint sur le type de retour

- Il est aussi possible d'indiquer le type de retour d'une méthode

```
class Operation:  
    def addition(self, a: int, b: int) -> int:  
        return a + b
```

- A la droite des paramètres, on ajoute une flèche avec le type de retour de la méthode.
- Là encore cela n'a pas de caractère contraignant mais cela rend les méthodes plus lisibles sur les types de données échangées avec l'extérieur.

TP n°2: Appels de méthodes d'instance



Les méthodes de classe

Python- Définition

- Une méthode de classe est une méthode qui **s'invoque sur le nom de la classe** elle-même
- Une méthode de classe se distingue par 2 choses :
 - Elle possède un premier paramètre appelé **cls** et qui représente la classe elle-même
 - Elle possède un décorateur **@classmethod**
- Exemple :

```
class Utilisateur:  
  
    statut = "SALARIE"  
  
    @classmethod  
    def change_statut(cls, statut):  
        cls.statut = statut
```

Python- Invocation d'une méthode de classe

- Contrairement à une méthode d'instance on ne crée pas d'objet pour invoquer la méthode.
- On invoque la méthode directement sur le nom de la classe.
- Exemple :

```
class Utilisateur:

    statut = "SALARIE"

    @classmethod
    def change_statut(cls, statut):
        cls.statut = statut

print(Utilisateur.statut)

Utilisateur.change_statut("Salarie")
print(Utilisateur.statut)
```

- Dans l'exemple ci-dessus la **méthode de classe change_statut** permet de changer la valeur de la **variable de classe statut**.

Python- Limitations d'une méthode de classe

- Une méthode de classe ne peut pas modifier une variable d'instance (exemple : `self.nom` ou `self.age`)
- Exemple :

```
class Utilisateur:

    statut = "SALARIE"

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    @classmethod
    def change_statut(cls, statut):
        cls.statut = statut
        self.age = 0
```

- L'IDE PyCharm détecte que l'utilisation de `self` est illégale ici.

Atelier (TP)

OBJECTIFS : Créer des attributs et méthodes de classes

DESCRIPTION :

- Dans le TP n°3 vous allez apprendre à mettre en place un attribut et une méthode de classe.



Les méthodes static

Python- Définition

- Une méthode static s'invoque également sur le nom de la classe elle-même
- Une méthode static se distingue par 2 choses :
 - Elle **ne possède pas** de premier paramètre technique
 - Elle possède un décorateur **@staticmethod**
- Exemple :

```
class Operation:  
  
    @staticmethod  
    def addition(val1, val2):  
        return val1 + val2  
  
resultat = Operation.addition(15, 25)  
print(resultat)
```

Python- méthode de classe vs méthode static

- Une **méthode static** ne peut pas modifier une variable de classe. Elle est indépendante de tout contexte
- Une **méthode static** est réservée aux **méthodes utilitaires**.

Atelier (TP)

OBJECTIFS : Créer des attributs et méthodes static

DESCRIPTION :

- Dans le TP n°4 vous allez apprendre à mettre en place une méthode static.



Encapsulation

Python - Définition

- Le concept d'**encapsulation** est propre à la programmation orientée objet
- L'encapsulation consiste en le **masquage** des méthodes et variables à usage interne dans une classe.
- S'il existe des moyens de masquer des attributs et méthodes d'instance dans la plupart des langages (mot clé private en Java, PHP et C# par exemple), rien de tel n'existe en Python.
- Il existe cependant des conventions permettant d'indiquer qu'une variable ou une méthode n'est pas publique et ne doit pas être accédée depuis l'extérieur de la classe.

Python- Visibilité classes et méthodes

- En POO, on distingue généralement **au moins** 2 niveaux de visibilité différents :
 - Les membres privés auxquels on ne peut accéder que depuis l'intérieur de la classe
 - Les membres publics auxquels on peut accéder depuis n'importe où.
- Ces niveaux de visibilité permettent de “protéger” certains membres de classes qui ne devraient pas être accédés dans n'importe quelle situation ou depuis n'importe quel endroit.

Python- Conventions

- En Python, des **conventions de nommage** servent à indiquer les **niveaux de visibilité**.
- Ces conventions sont les suivantes :
 - On préfixe les noms des membres qu'on souhaite définir comme "privés" avec deux underscores comme ceci : `__nom-du-membre` ;
 - On préfixe les noms des membres qu'on souhaite définir comme "protégés" avec un underscore comme ceci : `_nom-du-membre`.

Python- name mangling

- Python possède un mécanisme appelé **name mangling**
- Un membre de classe dont le nom commence par deux underscores, par exemple `__nom` sera remplacé lors de l'interprétation par `_Classe__nom`.
- Si un développeur essaie d'accéder un tel membre défini via `__nom`, Python renverra donc une erreur.

Python- Visibilité classes et méthodes

➤ Exemple:

```
class Animal:

    def infos(self):
        print("Je suis un animal")

    def __se_cacher(self):
        print("Je me cache pour me protéger")
```

```
an1 = Animal()
an1.__se_cacher()
```

```
Run: poo_name_mangling x
Traceback (most recent call last):
  File "C:\Users\RichardBONNAMY\PycharmProjects\preparationCours\poo_
    an1.__se_cacher()
AttributeError: 'Animal' object has no attribute '__se_cacher'
```

```
an1 = Animal()
an1._Animal__se_cacher()
```

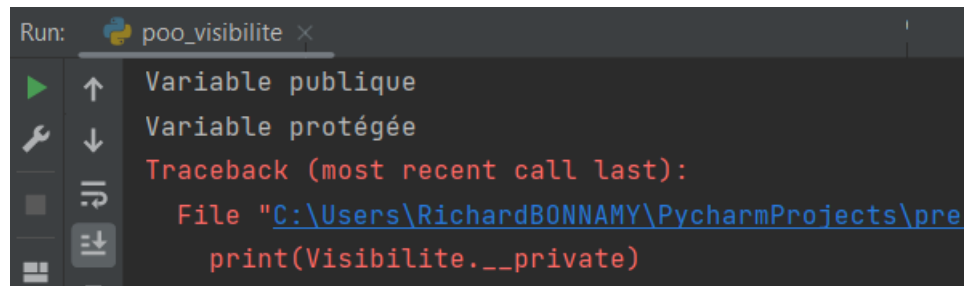
```
Run: poo_name_mangling x
Je me cache pour me protéger
```

Python- Visibilité

- Autre exemple :

```
class Visibilite:  
  
    public = "Variable publique"  
    _protected = "Variable protégée"  
    __private = "Variable privée"
```

```
print(Visibilite.public)  
print(Visibilite._protected)  
print(Visibilite.__private)
```



```
Run: poo_visibilite x  
Variable publique  
Variable protégée  
Traceback (most recent call last):  
  File "C:\Users\RichardBONNAMY\PycharmProjects\pre  
    print(Visibilite.__private)
```




Getters et setters

Python- getters et setters

- Dans de nombreux langages, l'encapsulation des variables s'accompagnent de la mise en place de méthodes appelées getters et setters.
- On peut retrouver cela aussi en Python :

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def get_age(self):
        return self.age

    def set_age(self, age):
        self.age = age

    def get_nom(self):
        return self.nom

    def set_nom(self, nom):
        self.nom = nom
```

```
ut1 = Utilisateur("Besos", 59)
ut2 = Utilisateur("Musk", 42)

ut1.set_nom("Bezos")
ut2.set_age(51)

print(ut1.get_nom()+" "+str(ut1.get_age()))
print(ut2.get_nom()+" "+str(ut2.get_age()))
```

Python- getters et setters

- En réalité ce n'est pas forcément le plus courant en Python. Ce n'est pas "Pythonique"
- On va plutôt utiliser des **properties**
- Ce sont des méthodes décorées particulières qui s'utilisent comme si on utilisait directement l'attribut

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    @property
    def age(self):
        return self.age

    @age.setter
    def age(self, value):
        self.age = value

    @property
    def nom(self):
        return self.nom

    @nom.setter
    def nom(self, value):
        print("appel setter nom")
        self.nom = value
```

```
ut1 = Utilisateur("Besos", 59)
ut2 = Utilisateur("Musk", 42)

ut1.nom = "Bezos"
ut2.age = 51

print(ut1.nom+" "+str(ut1.age))
print(ut2.nom+" "+str(ut2.age))
```

Ici on utilise les setters

Ici on utilise les getters

...mais il y a un mais...

Python- getters et setters

- Avec le code écrit tel quel, le code `self.nom = nom` appelle le setter qui s'appelle lui-même et on part en boucle infini.
- Pour l'illustrer j'ai ajouté un `print` dans le setter de `nom`

```
@nom.setter  
def nom(self, value):  
    print("appel setter nom")  
    self.nom = value
```

- Et voici ce qu'on obtient à l'exécution

```
appel setter nom  
appel setter nom  
appel setter nom  
appel setter nom  
appel setter nom
```

```
File "C:\Users\RichardBONNAMY\PycharmProjects\preparationCours\poo_methodes_pr  
    print("appel setter nom")  
RecursionError: maximum recursion depth exceeded while calling a Python object
```

- Comment faire ?

Python- getters et setters

- La solution est de renommer les attributs d'instance avec un _ devant :

```
class Utilisateur:

    def __init__(self, nom, age):
        self._nom = nom
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        self._age = age

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, nom):
        self._nom = nom
```

```
ut1 = Utilisateur("Besos", 59)
ut2 = Utilisateur("Musk", 42)

ut1.nom = "Bezos"
ut2.age = 51

print(ut1.nom+" "+str(ut1.age))
print(ut2.nom+" "+str(ut2.age))
```

Ici on utilise les setters

Ici on utilise les getters

Et là c'est "Pythonique" !

Atelier (TP)

OBJECTIFS : Mettre en place l'encapsulation des variables

DESCRIPTION :

- Dans le TP n°5 vous allez apprendre à mettre en place les règles de l'encapsulation via les propriétés (décorateurs).



Héritage

Python- Les Classes – Héritage en POO

- En programmation orientée objet, **l'héritage** est un mécanisme qui lie des classes entre elles
- Le lien d'héritage est de type **parent / enfant**.
- Par ce mécanisme une **classe parente** (classe mère ou super classe) va transmettre ses caractéristiques et méthodes à ses **filles (sous classes)**.
- C'est un **mécanisme de transmission**

Python- Les Classes – Héritage en Python

- Dans le langage Python le lien mère/fille doit être précisé au niveau de chaque fille. Le **nom de la mère est précisé entre parenthèses** dans la signature de la classe fille :

```
class Vehicule:
    def infos(self):
        print("Je suis un véhicule")
class Camion(Vehicule):
    pass
```

- *Le mot clé **pass** est un mot clé permettant d'indiquer que la définition de la classe est vide et permet d'éviter une erreur d'exécution.*

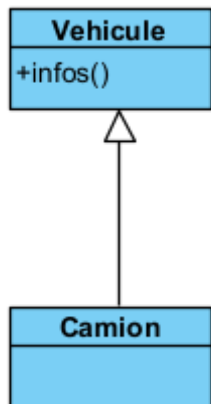
Python- Les Classes – Héritage bonnes pratiques

- Attention, **un héritage doit avoir du sens conceptuellement**.
- Python n'interdit pas de mettre en place de l'héritage entre n'importe quelle classe mais c'est une très mauvaise pratique.
- **Comment savoir si le lien a du sens ?** Si on peut dire "**est aussi**" ou "**est un type particulier de**" alors on a le droit de mettre en place un lien d'héritage.
- Peut-on dire qu'un Camion "est aussi" un Véhicule ? Peut-on dire que Camion "est un type particulier de" Véhicule ?

```
class Vehicule:  
  
    def infos(self):  
        print("Je suis un véhicule")  
  
class Camion(Vehicule):  
    pass
```

Python- Héritage et UML

- En UML, une flèche avec un triangle blanc vide formalise une relation d'héritage entre 2 classes
- La flèche pointe vers la classe mère.
- En UML on l'appelle généralisation, i.e. la classe Véhicule est une généralisation du Camion
- Dans l'autre sens on parle de spécialisation. Le Camion est une spécialisation du Véhicule.



```
class Vehicule:

    def infos(self):
        print("Je suis un véhicule")

class Camion(Vehicule):
    pass
```

Python- Les Classes – Héritage démonstration

- Dans l'exemple ci-dessous je crée un **Camion** nommé **c1** puis j'appelle la méthode **infos()**.
- Comme on le voit la méthode **infos()** appartient à la classe **Véhicule**

```
class Vehicule:  
    def infos(self):  
        print("Je suis un véhicule")  
  
class Camion(Vehicule):  
    pass
```

```
c1 = Camion()  
c1.infos()
```

```
Run: poo_heritage x  
▶ ↑ Je suis un véhicule
```

Python- Les Classes – Héritage et méthode `__init__`

- La classe mère peut posséder une méthode `__init__`
- Exemple :

```
class Vehicule:

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

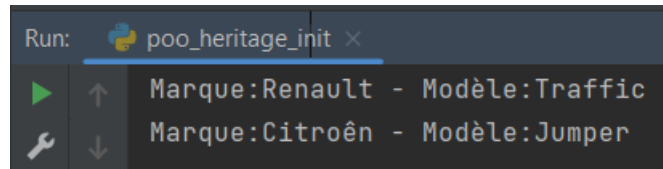
    def infos(self):
        print("Marque:" + self.marque + " - Modèle:" + self.modele)
```

```
class Camion(Vehicule):
    pass
```

- Je peux créer des instances de Camion en utilisant cette méthode `__init__` :

```
c1 = Camion("Renault", "Traffic")
c2 = Camion("Citroën", "Jumper")

c1.infos()
c2.infos()
```



```
Run: poo_heritage_init x
Marque:Renault - Modèle:Traffic
Marque:Citroën - Modèle:Jumper
```

Python- Les Classes – Héritage et méthode `__init__`

- La classe fille peut également posséder une méthode `__init__`
- Dans ce cas l'instanciation d'un Camion utilisera la méthode `__init__` de la classe fille et non celle de la classe mère
- On appelle cela une redéfinition de la méthode `__init__`

```
class Vehicule:

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def infos(self):
        print("Marque:" + self.marque + " - Modèle:" + self.modele)
```

```
class Camion(Vehicule):

    def __init__(self, marque, modele, volume):
        self.marque = marque
        self.modele = modele
        self.volume = volume
```

- Maintenant il n'est plus possible d'invoquer le constructeur avec 2 paramètres bien qu'il existe encore dans la classe mère. Le code ci-dessous échoue :

```
c1 = Camion("Renault", "Traffic")
c2 = Camion("Citroën", "Jumper")
```

```
Run: poo_heritage_init x
Traceback (most recent call last):
  File "C:\Users\RichardBONNAMY\PycharmProjects\preparationCours\poo_heritage_init.py", line 16, in <module>
    c1 = Camion("Renault", "Traffic")
TypeError: __init__() missing 1 required positional argument: 'volume'
```

Python- Les Classes – Problématique

- Problématique : on note que dans la classe fille et dans la classe mère, les **2 premières lignes des méthodes `__init__` sont identiques**.

```
class Vehicule:

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
```

```
class Camion(Vehicule):

    def __init__(self, marque, modele, volume):
        self.marque = marque
        self.modele = modele
        self.volume = volume
```

- **En POO on n'aime pas la duplication de code.**
- L'idéal serait donc que la **méthode `__init__` de la classe Camion invoque la méthode `__init__` de la classe Véhicule**.
- **Comment faire ?**

Python- Les Classes – Appel méthode `__init__` classe mère

- **Solution 1** : invocation de la méthode `__init__` de la classe mère de manière **statique**.
- **Exemple** :

```
class Vehicule:

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def infos(self):
        print("Marque:" + self.marque + " - Modèle:" + self.modele)
```

```
class Camion(Vehicule):

    def __init__(self, marque, modele, volume):
        Vehicule.__init__(self, marque, modele)
        self.volume = volume
```

- Comme vous le voyez on invoque la méthode `__init__` sur le nom de la classe mère.
- On passe bien évidemment les paramètres attendus.

Python- Les Classes – Utilisation super()

- **Solution 2 :** L'invocation dans l'exemple précédent est une mauvaise pratique
- **La bonne pratique est d'utiliser super()** qui est une référence à la super-classe et donc à la classe mère
- **Exemple :**

```
class Vehicule:

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def infos(self):
        print("Marque:"+self.marque+" - Modèle:"+self.modele)
```

```
class Camion(Vehicule):

    def __init__(self, marque, modele, volume):
        super().__init__(marque, modele)
        self.volume = volume
```

- Pourquoi est-ce une bonne pratique d'utiliser **super()** ? A cause de la MRO (Method Resolution Order) qui n'est mise en œuvre que si on utilise super().

Python- Les Classes – Redéfinition de méthode

- Vous voyez que si on invoque la **méthode infos()** sur une **instance de Camion**, l'**information de volume n'est pas affichée**.
- On **peut redéfinir la méthode infos()** pour la classe Camion

```
class Vehicule:  
  
    def __init__(self, marque, modele):  
        self.marque = marque  
        self.modele = modele  
  
    def infos(self):  
        print("Marque:"+self.marque+" - Modèle:"+self.modele)
```

```
class Camion(Vehicule):  
  
    def __init__(self, marque, modele, volume):  
        super().__init__(marque, modele)  
        self.volume = volume  
  
    def infos(self):  
        print("Marque:"+self.marque+" - Modèle:"+self.modele+" - Volume:"+str(self.volume))
```

Python- Les Classes – Redéfinition de méthode - exemple

```
class Vehicule:

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    def infos(self):
        print("Marque:" + self.marque + " - Modèle:" + self.modele)
```

```
class Camion(Vehicule):

    def __init__(self, marque, modele, volume):
        super().__init__(marque, modele)
        self.volume = volume

    def infos(self):
        print("Marque:" + self.marque + " - Modèle:" + self.modele + " - Volume:" + str(self.volume))
```

```
c1 = Camion("Renault", "Traffic", 11)
c2 = Camion("Citroën", "Jumper", 8)

c1.infos()
c2.infos()
```

```
Run: poo_heritage_init x
▶ ↑ Marque:Renault - Modèle:Traffic - Volume:11
⚙ ↓ Marque:Citroën - Modèle:Jumper - Volume:8
```

Atelier (TP)

OBJECTIFS : Mettre en place un héritage et une redéfinition de méthode

DESCRIPTION :

- Dans le TP n°6 vous allez apprendre à mettre en place un héritage entre 2 classes et à redéfinir dans la classe fille une méthode existant dans la classe mère



Héritage Compléments

Python- Héritage isinstance()

- Il est possible de **vérifier si une instance est d'une classe donnée ou non**.
- Pour cela on utilise la fonction **isinstance()** qui retourne True ou False
- La fonction **isinstance()** prend 2 paramètres:
 - L'instance à tester
 - Le nom de la classe à tester
- Exemple :

```
c1 = Camion("Renault", "Traffic", 11)
resultat_test = isinstance(c1, Camion)

print(resultat_test)
```

Run: poo_heritage_init x
True

```
c1 = Camion("Renault", "Traffic", 11)
resultat_test = isinstance(c1, Vehicule)

print(resultat_test)
```

Run: poo_heritage_init x
True


- Dans l'exemple ci-dessus on note que c1 est bien une instance de Camion et donc par héritage de Vehicule aussi.



Python- Héritage `issubclass()`

- Il est possible de **vérifier si une classe est une sous-classe d'une autre classe ou non**.
- Pour cela on utilise la fonction **`issubclass()`** qui retourne `True` ou `False`
- La fonction **`issubclass()`** prend 2 paramètres:
 - Le nom de la classe fille
 - Le nom de la classe mère
- Exemple :

```
resultat_test = issubclass(Camion, Vehicule)


print(resultat_test)
```



Run:  poo_heritage_init x

  True

```
resultat_test = issubclass(Vehicule, Camion)

print(resultat_test)
```

Run:  poo_heritage_issubclass x

  False

- Dans l'exemple ci-dessus on note que `Camion` est bien une sous-classe de `Vehicule` mais que l'inverse n'est pas vrai.

Python- Les Classes - Polymorphisme

- **Polymorphisme** signifie littéralement “plusieurs formes”.
- Il existe **plusieurs types de polymorphisme**. En général en POO le polymorphisme fait référence au polymorphisme par sous-typage dans le contexte de l'héritage.
- **Polymorphisme par sous-typage** : Une méthode peut prendre plusieurs formes. Redéfinition d'une méthode de la classe mère dans les classes filles.

```
class Animal:
    def infos(self):
        print("Je suis un animal")

class Mammifere(Animal):
    def infos(self):
        print("Je suis un mammifère")

class Insecte(Animal):
    def infos(self):
        print("Je suis un insecte")
```

```
a = Animal()
a.infos()
```

```
Run: poo_polymorphisme x
Je suis un animal
```

```
a = Mammifere()
a.infos()
```

```
Run: poo_polymorphisme x
Je suis un mammifère
```

```
a = Insecte()
a.infos()
```

```
Run: poo_polymorphisme x
Je suis un mammifère
```


Python- Les Classes - Polymorphisme

- **Conséquence de l'héritage et du polymorphisme.**
- Soit la classe Afficheur qui possède une méthode static afficher.
- Cette méthode prend un animal en paramètre et invoque la méthode infos() de l'animal.

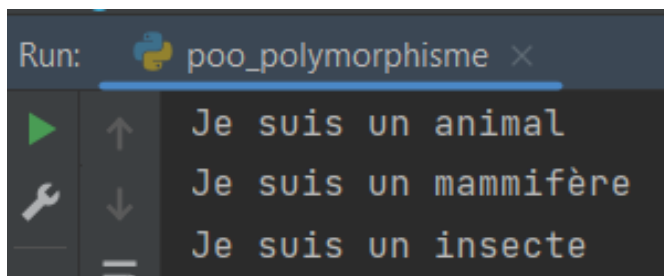
```
class Afficheur:  
  
    @staticmethod  
    def afficher(animal: Animal):  
        print(animal.infos())
```

```
a = Animal()  
b = Mammifere()  
c = Insecte()  
Afficheur.afficher(a)  
Afficheur.afficher(b)  
Afficheur.afficher(c)
```

- **Sachant que la méthode afficher prend un animal en paramètre, est-ce que ce code s'exécute sans erreur ?**
- **Qu'affiche cette méthode pour a, b et c ?**

Python- Les Classes - Polymorphisme

- **Résultat :**
- Ce code s'exécute sans erreur car bien que la méthode afficher prenne un animal en paramètre, n'oublions pas qu'un mammifère est aussi un animal, tout comme l'insecte.
- De plus Python sait quel type d'objet lui est passé en paramètre et invoque la bonne méthode : soit celle d'**Animal** pour **a**, soit celle de **Mammifère** pour **b**, soit celle d'**Insecte** pour **c**.



```
Run: poo_polymorphisme x
Je suis un animal
Je suis un mammifère
Je suis un insecte
```

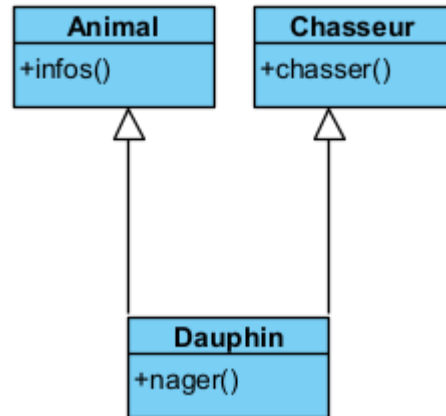
- **Conséquence : pensez à développer des méthodes génériques s'appliquant à des familles d'objet.**

Python- Les Classes – Héritage multiple

Héritage multiple

- Python gère l'héritage multiple.
- On parle d'héritage multiple en programmation orientée objet lorsqu'une sous-classe peut hériter de plusieurs classes mères différentes.
- Dans la pratique, l'héritage multiple est une chose très difficile à mettre en place
 - Exemple : cas où plusieurs classes mères définissent les mêmes variables et fonctions.

➤ **Exemple :**



- Ici Dauphin hérite d'Animal et de Chasseur

```
class Animal:

    def infos(self):
        print("Je suis un animal")

class Chasseur:

    def chasser(self):
        print("Je chasse en groupe")

class Dauphin(Animal, Chasseur):

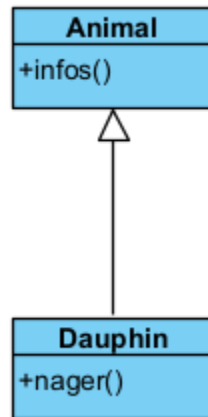
    def nager(self):
        print("Je sais nager")
```

Python- MRO

➤ MRO: Method Resolution Order

- Si vous invoquez une méthode sur une classe fille, Python recherche dans la classe fille cette méthode. S'il ne la trouve pas dans la classe fille il remonte dans la classe mère et ainsi de suite.

➤ Exemple :



```
class Animal:
    def infos(self):
        print("Je suis un animal")

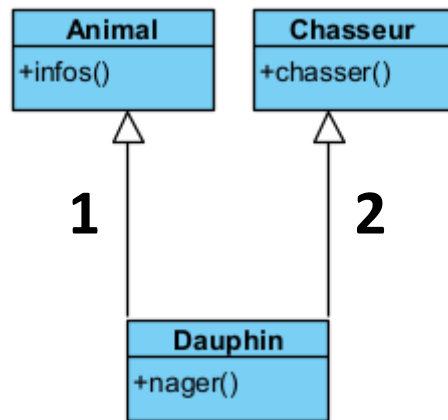
class Dauphin(Animal):
    def nager(self):
        print("Je sais nager")
```

```
dauphin1 = Dauphin()
dauphin1.infos()
dauphin1.nager()
```

- Pour la méthode **infos()** Python regarde dans la classe Dauphin, mais ne la trouve pas, puis remonte dans Animal et la trouve
- Pour la méthode **nager()** Python regarde dans Dauphin et la trouve

Python- MRO – Héritage multiple

- Dans le cas de l'héritage multiple, le **MRO** se fait dans l'ordre de déclaration de l'héritage
- **Exemple :**



```
dauphin1 = Dauphin()
dauphin1.infos()
dauphin1.nager()
dauphin1.chasser()
```

```
class Animal:

    def infos(self):
        print("Je suis un animal")

class Chasseur:

    def chasser(self):
        print("Je chasse en groupe")

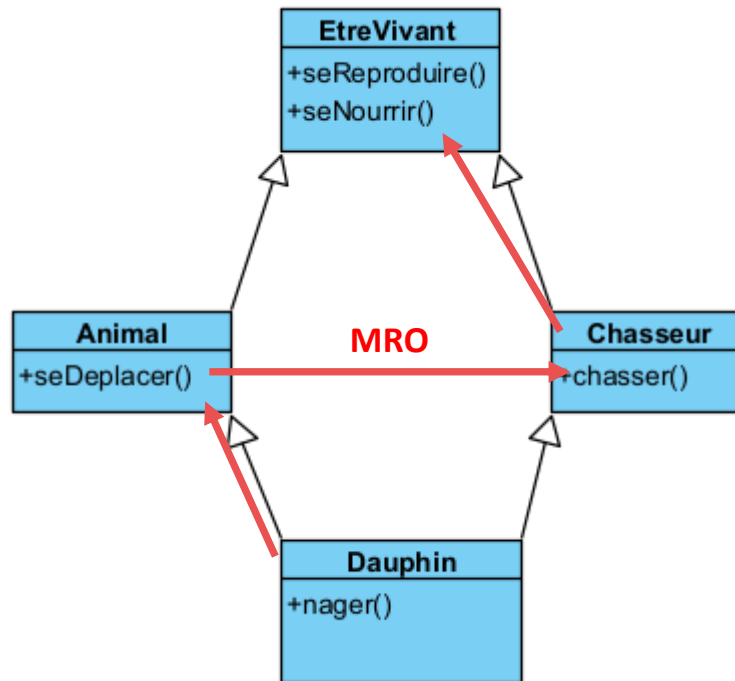
class Dauphin(Animal, Chasseur):

    def nager(self):
        print("Je sais nager")
```

- Dauphin hérite d'abord de Animal, puis de Chasseur
- Pour la méthode **chasser()** Python regarde d'abord dans Dauphin, puis Animal, puis Chasseur

Python- MRO – Formation en diamant

- Dans le cas d'une formation en diamant, la **MRO** se fait dans l'ordre des flèches rouges :
- **Exemple :**



- Si vous avez utilisé **super()** pour les appels par exemple de `__init__` la MRO sera bien utilisée et la méthode `__init__` de la classe **EtreVivant** ne sera appelée qu'une seule fois
- Si vous avez utilisé l'appel static pour les appels de la **méthode `__init__`** alors celle de la classe **EtreVivant** sera appelée 2 fois.



Classes et méthode abstraites

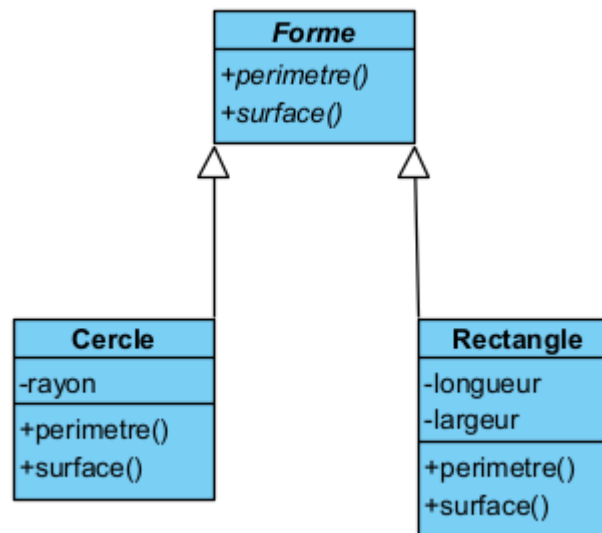
Python- Abstraction - problématique

Classe Forme :

- On a créé une classe Forme pour factoriser attributs et méthodes en commun
- On souhaite imposer aux classes filles des **méthodes obligatoires**.

Dans l'exemple ci-dessous, chaque classe fille doit obligatoirement posséder les méthodes `périmètre()` et `surface()`.

Comment faire ?



Python- Abstraction - problématique

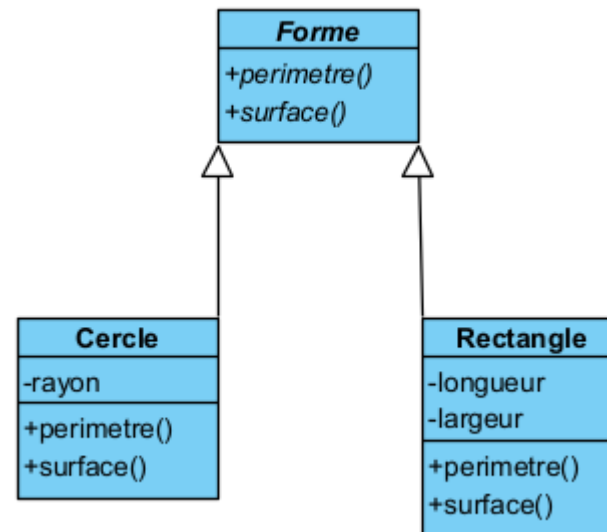
Autre problématique :

- Quel code contiennent les méthodes `périmètre()` et `surface()` de la classe `Forme` ?
- Peut-on fournir un code générique dans ces méthodes convenant à toutes les classes filles ?

Réponse :

- Elles ne peuvent pas contenir de code car aucun code générique ne peut convenir à l'ensemble des classes filles.

On va donc les rendre **abstraites**.



Python- Abstraction - solution

La **classe Forme** est également **abstraite**. Elle ne peut plus être instanciée.

Implémentation en Python :

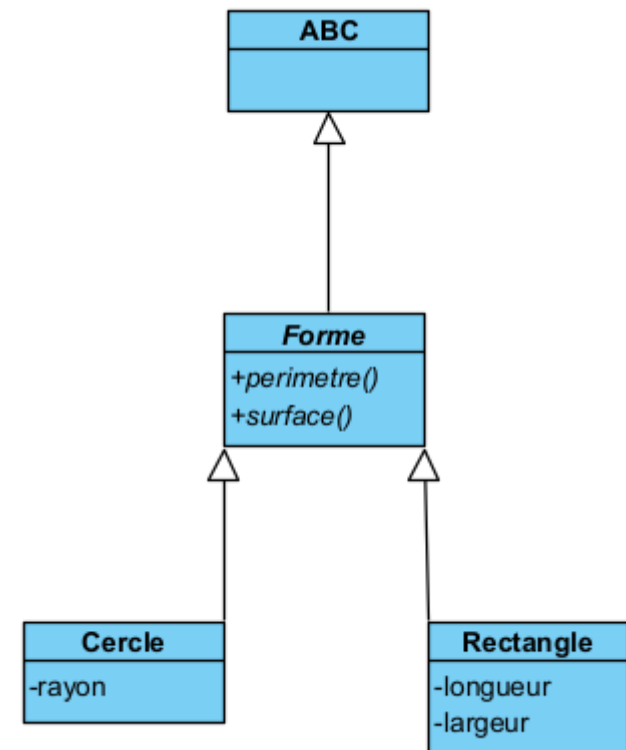
- import du module ABC (Abstract Base Classes)
- Forme hérite de la classe ABC
- les méthodes abstraites reçoivent le décorateur

@abstractmethod

```
from abc import ABC, abstractmethod

class Forme(ABC):
    @abstractmethod
    def perimetre(self):
        pass

    @abstractmethod
    def surface(self):
        pass
```



Python- Abstraction - problématique

```
from math import pi
from abc import ABC, abstractmethod

class Forme(ABC):
    @abstractmethod
    def perimetre(self):
        pass

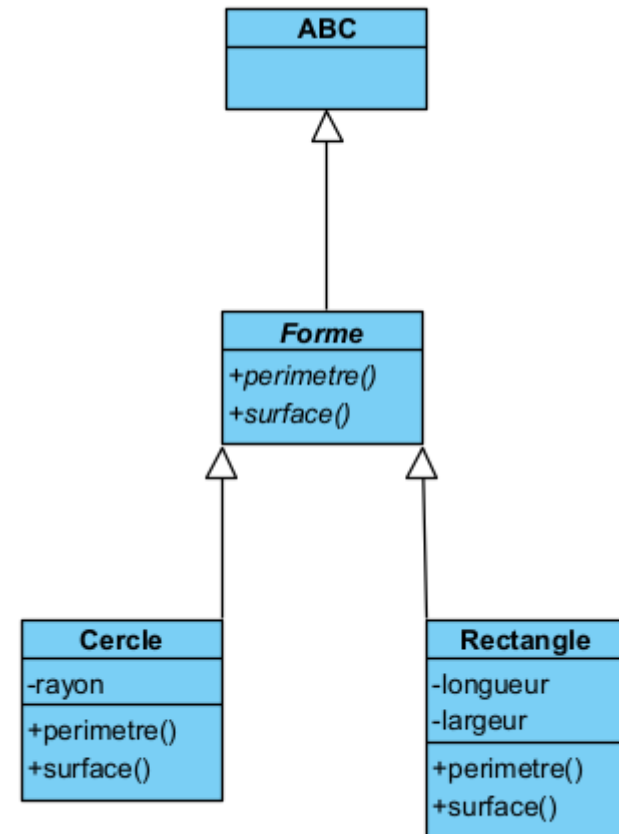
    @abstractmethod
    def surface(self):
        pass

class Cercle(Forme):

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

    def perimetre(self):
        return 2.0*pi*self.rayon

    def surface(self):
        return pi*self.rayon**2
```



Atelier (TP)

OBJECTIFS : Mettre en place un héritage avec de l'abstraction

DESCRIPTION :

- Dans le TP n°7 vous allez apprendre à mettre en place une classe abstraite contenant des méthodes abstraites et à les redéfinir dans les classes filles.



TP mise en oeuvre

Atelier (TP)

OBJECTIFS : Mettre en œuvre les concepts vus jusqu'à présent

DESCRIPTION :

- Dans le TP n°8 vous allez mettre en œuvre tous les concepts vus jusqu'à présent dans le cadre d'une dizaine d'exercices.



Les tuples

Python- Présentation

- Les **tuples** sont un autre type **séquentiel** de données.
- Les **tuples** ressemblent aux listes : un tuple consiste en différentes valeurs séparées par des virgules.
- On encadre généralement les valeurs d'un **tuple** avec un **couple de parenthèses** même si cela n'est **pas obligatoire**.
- Les **tuples** peuvent contenir différents types de valeurs comme des nombres, des chaines, des listes etc. et même d'autres tuples imbriqués.

```
# Données homogènes
t1 = 1, 2, 3

# Données hétérogènes
t2 = "Jane", 19

# Déclaration avec les ()
t3 = (3.1415, 34, "Data")

# Tuple contenant un tuple
t4 = "Jane", 19, (3.1415, 34, "Data")
```


Python- Accès aux données d'un tuple

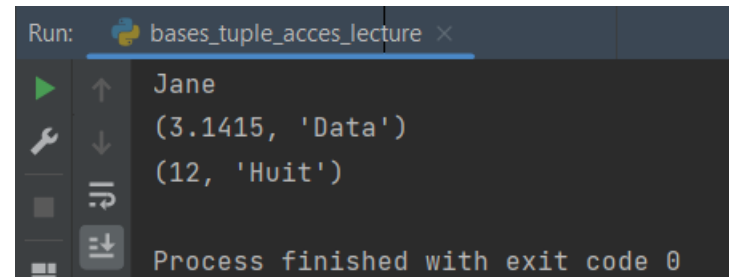
- Un **tuple** n'est accessible qu'en **lecture**. Un tuple est **immuable**.
- Le **tuple** est **indexé**
- L'accès à une donnée **d'index n** se fait via l'opérateur **crochet** : *mon_tuple* [n]
- Il est possible d'extraire un **sous-tuple** d'un tuple avec une tranche : *nom_tuple* [min:max]
- La fonction native **len(mon_tuple)** permet de connaître la **taille d'un tuple**
- Le **dernier élément d'un tuple** a pour index **len(mon_tuple)-1**

```
# Déclaration du tuple
t1 = "Jane", 19, 3.1415, "Data", (12, "Huit")

# Affichage de l'élément d'index 1
print(t1[0])

# Affichage de la tranche entre 2 et 4 (max exclu)
print(t1[2:4])

# Affichage du dernier élément
print(t1[len(t1)-1])
```



```
Run: bases_tuple_acces_lecture x
Jane
(3.1415, 'Data')
(12, 'Huit')
Process finished with exit code 0
```

Python- Tuple vide ou à valeur unique

- Pour créer un **tuple vide**, on utilisera une paire de parenthèses vides.
- Si on souhaite créer un tuple avec une seule valeur, alors il faudra faire suivre cette valeur d'une virgule.

```
tuple_vide = ()  
print(tuple_vide, len(tuple_vide))  
  
tuple_unique = ("Pierre", )  
print(tuple_unique, len(tuple_unique))
```

- Important :
 - Si on omet la virgule pour créer la variable **tuple_unique**, cette dernière ne sera pas considérée comme un tuple mais comme une string.

Python- Déballage

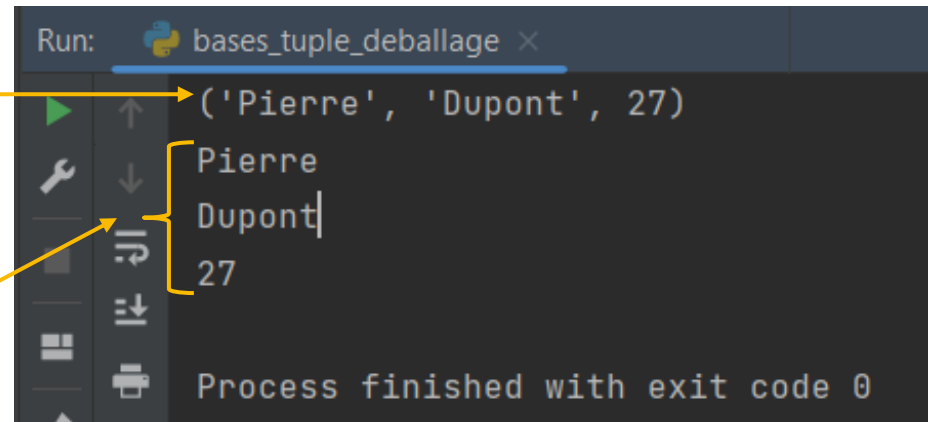
➤ Le déballage d'un tuple

- Un déballage correspond à une façon rapide d'affecter les différentes valeurs d'un tuple dans des variables séparées.

```
seq = ("Pierre", "Dupont", 27)
print(seq)

# Déballage du tuple
nom, prenom, age = seq

print(nom)
print(prenom)
print(age)
```



Run: bases_tuple_deballage x

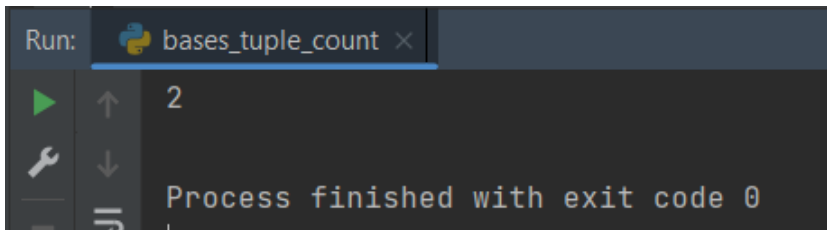
```
('Pierre', 'Dupont', 27)
Pierre
Dupont
27
Process finished with exit code 0
```

Python- La méthode count(e)

- Les **tuples** possèdent la méthode `count(elt)` où `elt` représente l'élément dont on veut compter le nombre d'occurrences.
- Exemples pour compter le nombre de fois où le prénom "Pierre" apparaît dans le tuple :

```
# Déclaration du tuple
prenoms = "Khalid", "Laurence", "Pierre", "Axel", "Pierre", "Samara"

# Comptage
nb = prenoms.count("Pierre")
print(nb)
```



```
Run: bases_tuple_count x
2
Process finished with exit code 0
```



Les dictionnaires

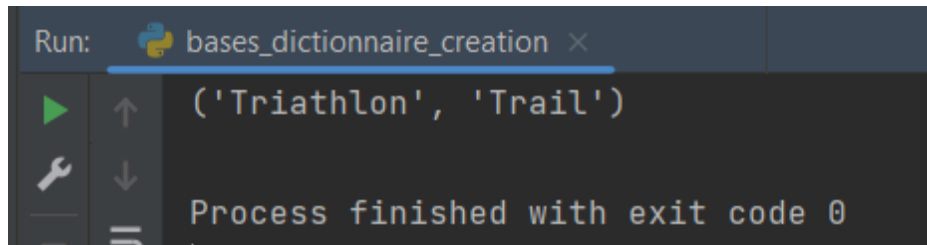
Python- Présentation

- Les dictionnaires sont un type natif de données Python.
- Ce type de données peut, de la même façon que les données séquentielles, contenir plusieurs valeurs et chaque valeur va être "indexée".
- A la différence des listes, les **dictionnaires** utilisent une **clé de stockage** libre.
- On peut par exemple utiliser une chaîne de caractère, un tuple ou n'importe quel objet comme clé de stockage.
- On peut également utiliser un nombre si besoin, comme un identifiant.

Python- Création

- Pour créer un nouveau dictionnaire, nous allons devoir utiliser un couple d'accolades { }
- On définit ensuite des paires **clef : valeur** à l'intérieur des accolades.
- Les paires sont séparées par des virgules

```
dico = {"nom": "DUPONT",  
        "prenom": "Pierre",  
        "sport": ("Triathlon", "Trail")}  
print(dico["sport"])
```



Run: bases_dictionnaire_creation ×

▶ ('Triathlon', 'Trail')

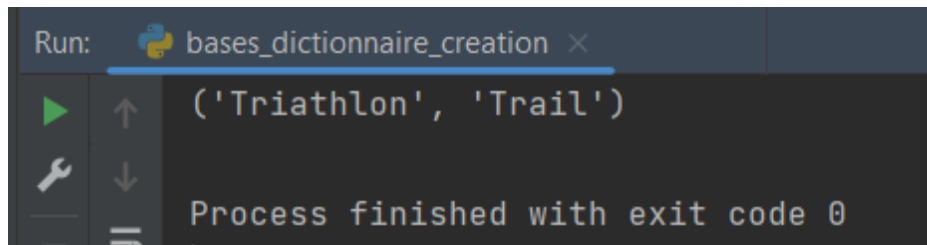
⚙

Process finished with exit code 0

Python- Extraire une valeur

- L'extraction d'une valeur utilise l'opérateur crochet [] avec le nom de la clef.
- Exemple ci-dessous :

```
dico = {"nom": "DUPONT",  
        "prenom": "Pierre",  
        "sport": ("Triathlon", "Trail")}  
print(dico["sport"])
```



```
Run: bases_dictionnaire_creation ×  
('Triathlon', 'Trail')  
Process finished with exit code 0
```


Python- Modification d'une valeur

- On utilise l'**opérateur []** avec le nom de la clé à modifier..
- *nom_dico* [**nom_cle**]=nouvelle_valeur
- La nouvelle valeur remplace l'ancienne
- **Exemple:**

```
dico = {"nom": "DUPONT", "prenom": "Pierre", "sport": ("Triathlon", "Trail")}
print(dico)

dico["nom"] = "DUPONTEL"
print(dico)
```

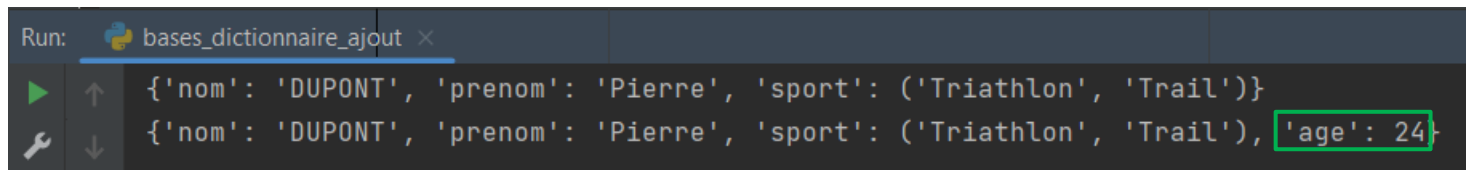
Run: bases_dictionnaire_modification ×

```
{'nom': 'DUPONT', 'prenom': 'Pierre', 'sport': ('Triathlon', 'Trail')}
{'nom': 'DUPONTEL', 'prenom': 'Pierre', 'sport': ('Triathlon', 'Trail')}
```

Python- Ajout d'une valeur

- On utilise l'**opérateur []** avec le nom de la clé à ajouter.
- La clé ne doit pas exister
- *nom_dico* [***nouvelle_cle***] = valeur
- La nouvelle valeur remplace l'ancienne
- **Exemple:**

```
dico = {"nom": "DUPONT", "prenom": "Pierre", "sport": ("Triathlon", "Trail")}\nprint(dico)\n\ndico["age"] = 24\nprint(dico)
```

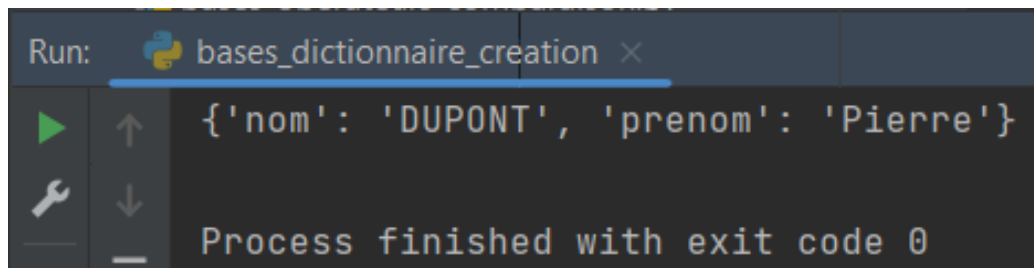


```
Run: bases_dictionnaire_ajout x\n\n▶ ↑ {'nom': 'DUPONT', 'prenom': 'Pierre', 'sport': ('Triathlon', 'Trail')}\n⚙ ↓ {'nom': 'DUPONT', 'prenom': 'Pierre', 'sport': ('Triathlon', 'Trail'), 'age': 24}
```

Python- Suppression d'une valeur

- Pour supprimer une entrée dans le dictionnaire, on utilise l'instruction **del** suivi du nom du dictionnaire avec la clef de l'élément à supprimer entre crochets comme ceci :
- **del nom_dico [nom_cle]**
- **Exemple :**

```
dico = {"nom": "DUPONT",  
        "prenom": "Pierre",  
        "sport": ("Triathlon", "Trail")}  
  
del dico["sport"]  
print(dico)
```

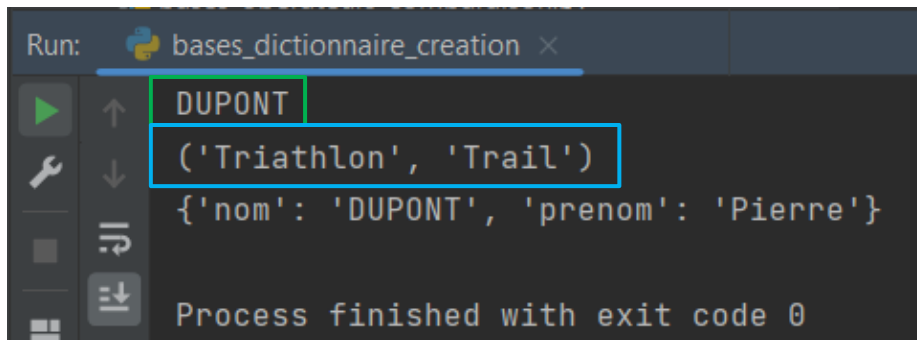


```
Run: bases_dictionnaire_creation ×  
▶ {'nom': 'DUPONT', 'prenom': 'Pierre'}  
⚙️  
= Process finished with exit code 0
```

Python- Dictionnaires - Les méthodes

- Les dictionnaires possèdent des méthodes notamment `get()` et `pop()`
- `nom_dico.get(nom_cle)` extrait la valeur de la clé *nom_cle* du dictionnaire *nom_dico*
- `pop(nom_cle)` extrait la valeur de la clé *nom_cle* et supprime l'entrée dans le dictionnaire *nom_dico*
- Exemple :

```
dico = {"nom": "DUPONT",  
        "prenom": "Pierre",  
        "sport": ("Triathlon", "Trail")}  
nom = dico.get("nom") # extrait la valeur pour la clé nom  
print(nom)  
  
sports = dico.pop("sport") # extrait puis supprime le dernier couple du dictionnaire  
print(sports)  
print(dico) # on note dans la console que le dico ne contient plus la clé sport
```








```
Run: bases_dictionnaire_creation x  
DUPONT  
(('Triathlon', 'Trail'))  
{'nom': 'DUPONT', 'prenom': 'Pierre'}  
Process finished with exit code 0
```

Python- Dictionnaires – keys() et values()

- La méthode **keys()** d'un dictionnaire retourne un objet de type **dict_keys**
- La méthode **values()** d'un dictionnaire retourne un objet de type **dict_values**
- Ce sont des **itérables** sur lesquels on ne peut faire ni append, ni pop, pour autant ces objets sont **mutables**
- Ce sont les modifications apportées sur le dictionnaire qui les modifient.
- **Exemple :**

```
inventaire = { "boulons":25,  
               "vis":42,  
               "perceuses":8,  
             }  
values = inventaire.values()  
print(values)  
  
inventaire["visseuse"]=45  
  
print(values)
```

Run:  tuples x

		dict_values([25, 42, 8])
		dict_values([25, 42, 8, 45])

Dictionnaires – parcours avec une boucle

- La méthode en passant par les **items**:

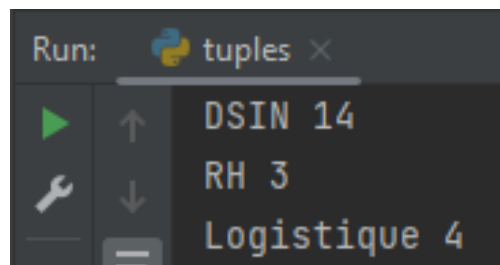
```
salaries = { "DSIN": 14, "RH": 3, "Logistique": 4}

for key, value in salaries.items():
    print(key, value)
```

- La méthode en passant par les **keys**:

```
salaries = { "DSIN": 14, "RH": 3, "Logistique": 4}

for key in salaries.keys():
    value = salaries.get(key)
    print(key, value)
```



Atelier (TP)

OBJECTIFS : Savoir utiliser un dictionnaire

DESCRIPTION :

- Dans le TP n°9 vous allez apprendre à manipuler un dictionnaire dans différents cas.



Les sets

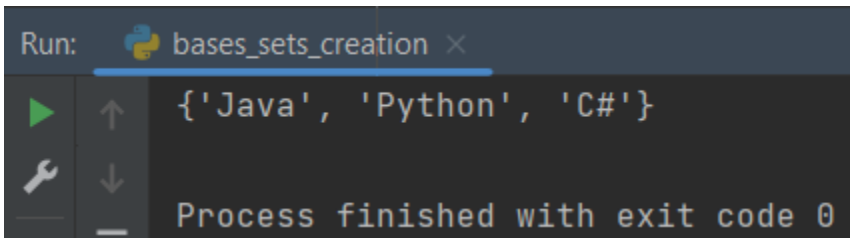
Python- Présentation

- Les **sets** forment un autre type de données composites.
- Un **set** est une collection "**non ordonnée**" d'éléments, **sans index** et qui ne contient **pas de doublon**.
- Une des utilisations les plus courantes des sets est de les utiliser pour supprimer des valeurs en doublon à partir d'un autre type de données.
- Pour créer un set, nous allons utiliser une paire d'accolades { } en plaçant les différents éléments de notre ensemble entre ces accolades en les séparant avec une virgule.
- Notez que pour créer un ensemble vide il faudra utiliser la fonction set() car la syntaxe { } va créer un dictionnaire vide et non pas un ensemble vide.

Python- Création

- Pour **créer un set**, nous allons utiliser l'opérateur accolades { } et placer les éléments entre ces accolades en les **séparant avec une virgule**.
- Un **set n'accepte pas les doublons**

```
ensemble = {"Java", "Python", "C#", "Python"}  
print(ensemble)
```



```
Run: bases_sets_creation ×  
▶ {'Java', 'Python', 'C#'}  
⚙ Process finished with exit code 0
```

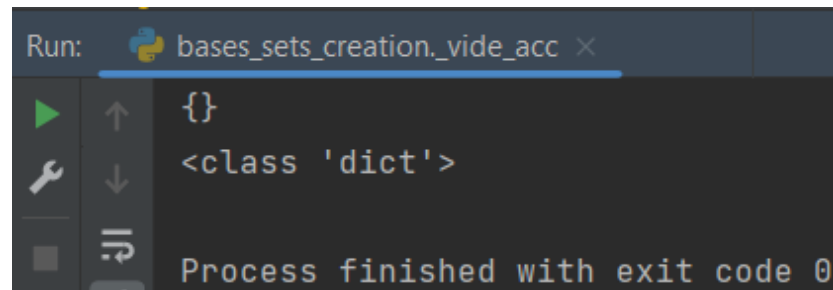
L'élément en doublon a été supprimé

- *NDLR : L'opérateur { } est aussi utilisé pour les dictionnaires sauf que pour les sets il n'y a pas de clé.*

Python- Création d'un set vide

- Notez que pour créer un ensemble vide il faudra utiliser la fonction **set()** car la syntaxe **{ }** va créer un **dictionnaire vide** et non pas un ensemble vide.
- **Exemple avec { } :**

```
ensemble = {}  
print(ensemble)  
print(type(ensemble))
```

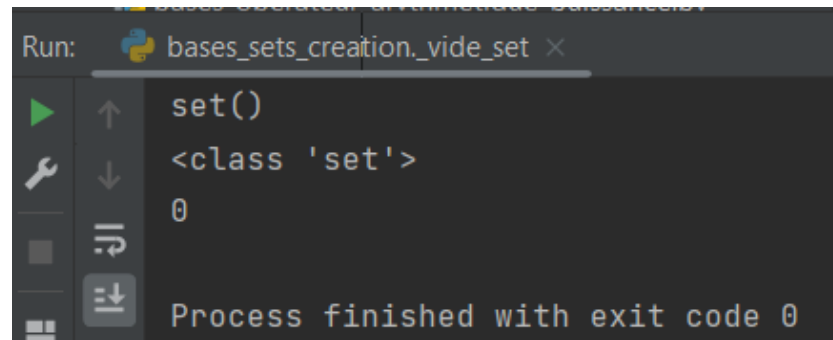


Run: bases_sets_creation._vide_acc x

```
{}  
<class 'dict'>  
  
Process finished with exit code 0
```

- **Exemple avec set():**

```
ensemble = set()  
print(ensemble)  
print(type(ensemble))  
print(len(ensemble))
```



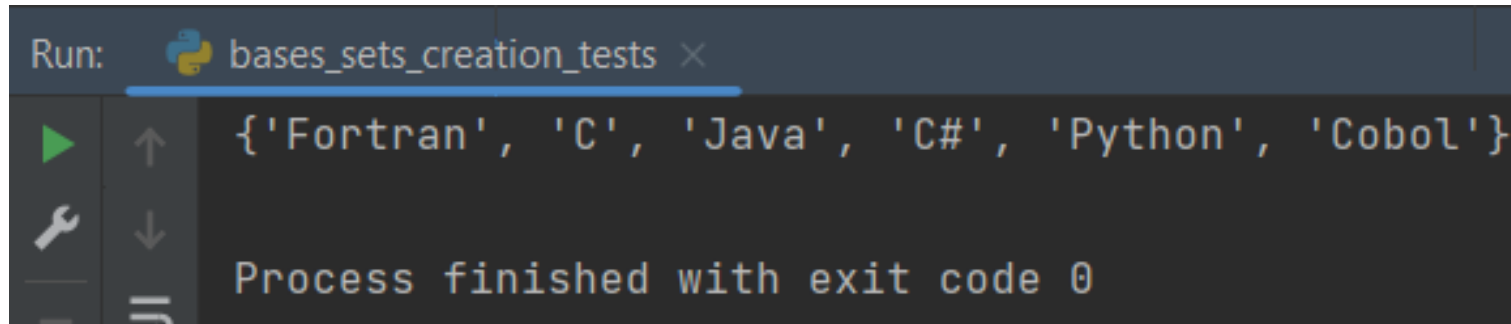
Run: bases_sets_creation._vide_set x

```
set()  
<class 'set'>  
0  
  
Process finished with exit code 0
```

Python- Ajout d'un élément

- Pour ajouter un élément, on utilise la méthode ***mon-set.add(element)***
- **Exemple :**

```
ensemble = {"Java", "Python", "C#", "Python", "C", "Cobol"}  
ensemble.add("Fortran")  
print(ensemble)
```



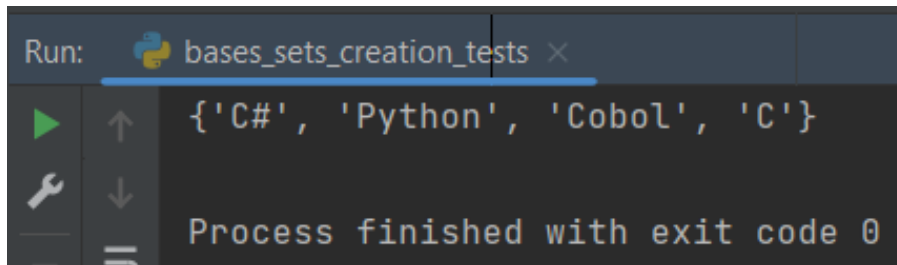
```
Run: bases_sets_creation_tests ×  
{'Fortran', 'C', 'Java', 'C#', 'Python', 'Cobol'}  
Process finished with exit code 0
```

Python- Suppression d'un élément

- Pour supprimer un élément, on utilise la méthode **discard(*element*)**

- **Exemple :**

```
ensemble = {"Java", "Python", "C#", "Python", "C", "Cobol"}  
ensemble.discard("Java")  
print(ensemble)
```

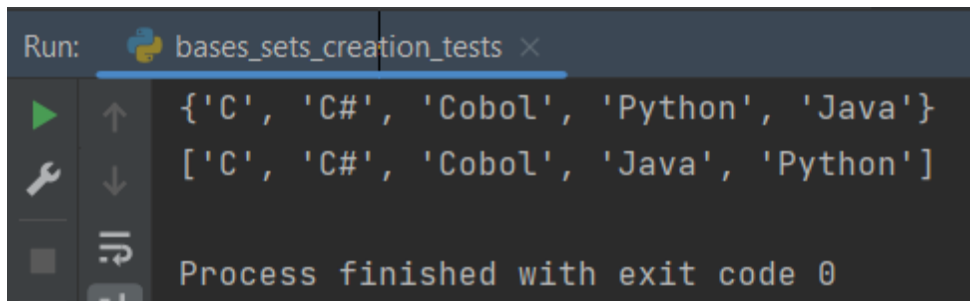


- Pour supprimer un élément, on peut aussi utiliser la méthode **remove(*element*)**
- Attention, à la différence de discard, **la méthode remove renvoie une exception si l'élément n'existe pas.**

Python- Le set n'est pas triable

- Un **set** stocke les objets selon leur hashcode. Le hashcode est un algorithme qui produit un entier pour une instance donnée.
- **Set et fonction native :**

```
ensemble = {"Java", "Python", "C#", "Python", "C", "Cobol"}  
print(ensemble)  
print(sorted(ensemble))
```



```
Run: bases_sets_creation_tests ×  
↑ {'C', 'C#', 'Cobol', 'Python', 'Java'}  
↓ ['C', 'C#', 'Cobol', 'Java', 'Python']  
Process finished with exit code 0
```

- La fonction native **sorted** ne tri pas le set mais retourne comme résultat une liste contenant les éléments triés.



Points clés sur les ensembles de données

Python- Points clés sur les ensembles

- Les types de données composite étudiés sont les **listes**, les **tuples**, les **dictionnaires** et les **sets**.
- Il est généralement peu évident de choisir quel type de données utiliser car “ils se ressemblent tous”.
- Voici donc un résumé des grandes caractéristiques de ces types et ce qui les différencie :
 - Les **listes** sont des collections **d'éléments ordonnés** et **altérables** qui peuvent contenir des doublons.
 - Les **tuples** sont des collections **d'éléments ordonnés** et **immuables** qui peuvent contenir des doublons.
 - Les **dictionnaires** sont des collections d'éléments **non ordonnés** mais **indexés avec des clefs** de notre choix et **altérables** qui n'acceptent pas de contenir plusieurs fois la même clef.
 - Les **sets** sont des collections d'éléments **non ordonnées, non indexés** et **altérables** qui n'acceptent pas les doublons.

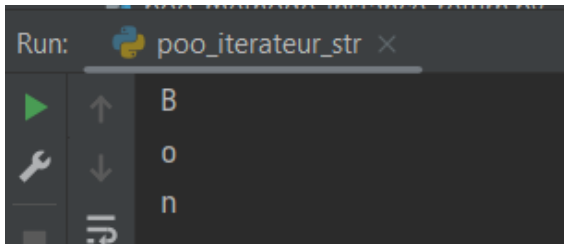


Itérateurs

Python- Définition

- Un **objet itérable** est un objet qu'on peut parcourir avec une **boucle for** ou avec une méthode **built-in** appelée **next()**
- Nous avons déjà utilisé des objets itérables fournis par Python comme les **listes**, **tuples** ou **string**
- Lorsque nous utilisons une **boucle for** pour parcourir un objet itérable, la boucle appelle en fait en interne à la fonction **iter()** sur l'objet itérable.
- Cette méthode **iter()** retourne un itérateur permettant de parcourir l'objet.
- Nous pouvons utiliser la fonction **iter()** pour faire comme la boucle for, i.e. récupérer un itérateur sur l'objet itérable.
- **Exemple :**

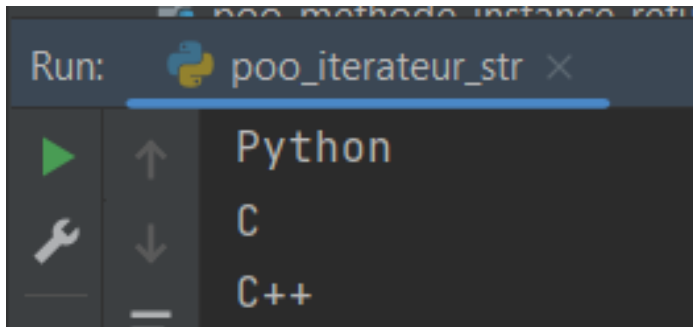
```
it1 = iter("Bonjour")  
print(next(it1))  
print(next(it1))  
print(next(it1))
```



Python- Itérateur sur une liste

➤ Exemple :

```
langages = ["Python", "C", "C++", "C#", "PHP", "Java"]  
it2 = iter(langages)  
print(next(it2))  
print(next(it2))  
print(next(it2))
```



Python- Créer une classe itérable

- On peut créer nos propres classes itérables.
- Pour cela, il suffit de définir une méthode `__iter__()` qui renvoie un objet disposant d'une méthode `__next__()`.
- Si la classe définit elle-même la méthode `__next__()`, alors `__iter__()` peut simplement renvoyer `self`.
- La méthode `__next__()` fait évoluer l'itération en interne

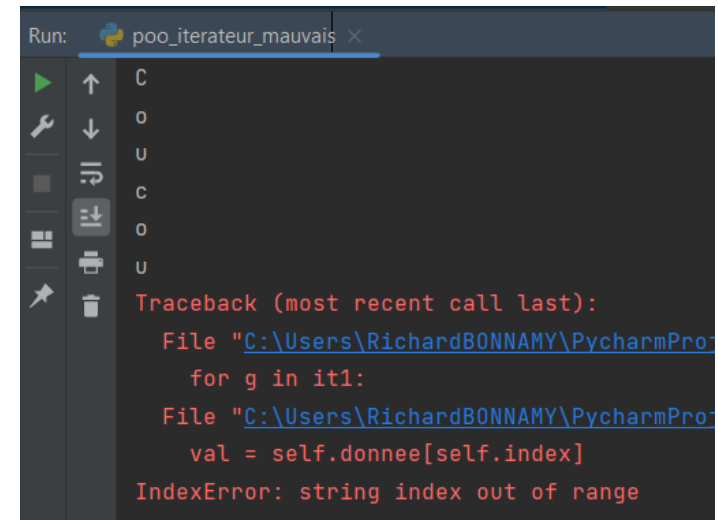
```
class Iterateur:  
  
    def __init__(self, index):  
        self.index = index  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        self.index += 1  
        return self.index
```

Python- Stopper une itération

- La méthode `__next__()` doit être capable de signifier, par exemple à une boucle `for`, qu'on est arrivé en fin d'itération, sinon une exception se produit
- Exemple :

```
class Iterateur:  
  
    def __init__(self, donnee):  
        self.donnee = donnee  
        self.index = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        val = self.donnee[self.index]  
        self.index += 1  
        return val
```

```
it1 = Iterateur("Coucou")  
for g in it1:  
    print(g)
```



```
Run: poo_iterateur_mauvais x  
C  
o  
u  
c  
o  
u  
  
Traceback (most recent call last):  
  File "C:\Users\RichardBONNAMY\PycharmPro  
    for g in it1:  
  File "C:\Users\RichardBONNAMY\PycharmPro  
    val = self.donnee[self.index]  
IndexError: string index out of range
```

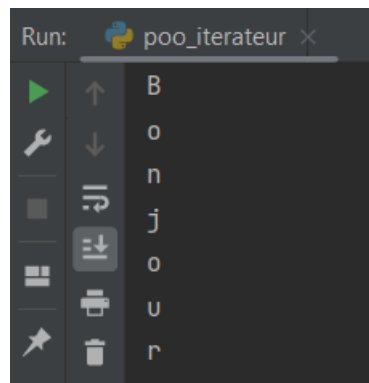
- Dans cet exemple la méthode `__next__()` retourne le caractère suivant d'une chaîne de caractère sans aucun contrôle pour savoir si on est arrivé en bout de chaîne.
- La boucle `for` s'exécute jusqu'à exception

Python- Stopper une itération avec StopIteration

- La méthode `__next()` doit lancer une exception appelée **StopIteration** pour signifier à la boucle for de s'arrêter.
- Cette exception est masquée à l'exécution.
- Exemple :

```
def __next__(self):  
    if self.index == len(self.donnee):  
        raise StopIteration  
    else:  
        val = self.donnee[self.index]  
        self.index += 1  
        return val
```

```
it1 = Iterateur("Coucou")  
for g in it1:  
    print(g)
```

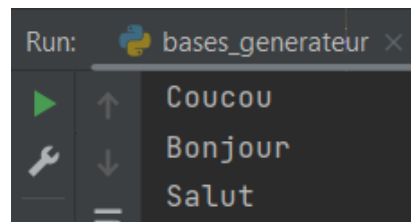


Python- Générateur – Définition

- Un **générateur** permet de **créer un itérateur**.
- Une **fonction** de **type générateur** utilise l'**instruction yield** à la place de return pour retourner des données.
- L'instruction **yield** met en pause le contexte d'exécution de la fonction jusqu'à l'appel suivant.
- Lorsqu'on définit un générateur, les méthodes `__iter__()` et `__next__()` sont créées automatiquement.
- **Exemple 1 :**

```
def generateur():  
    yield "Coucou"  
    yield "Bonjour"  
    yield "Salut"
```

```
gen = generateur()  
print(next(gen))  
print(next(gen))  
print(next(gen))
```

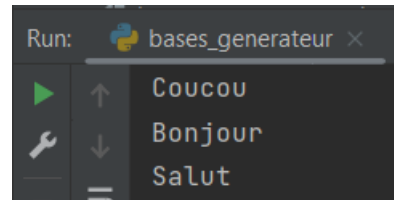


```
Run: bases_generateur x  
Coucou  
Bonjour  
Salut
```

Python- Exemples

➤ Exemple 2 :

```
def generateur():  
    yield "Coucou"  
    yield "Bonjour"  
    yield "Salut"  
  
for elt in generateur():  
    print(elt)
```

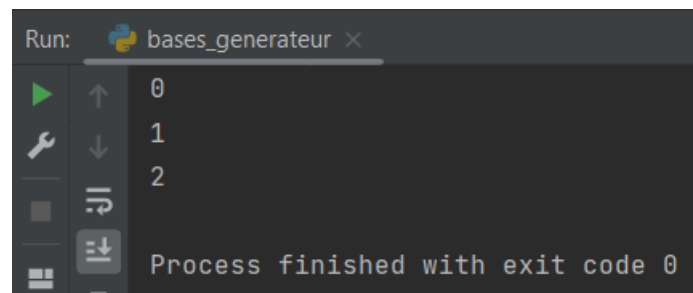


Run: bases_generateur x

Coucou
Bonjour
Salut

➤ Exemple 3:

```
def generateur2():  
    for x in range(0, 100):  
        yield x  
  
gen2 = generateur2()  
print(next(gen2))  
print(next(gen2))  
print(next(gen2))
```



Run: bases_generateur x

0
1
2
Process finished with exit code 0



`__str__ et
__repr__`

Python- Affichage d'une instance

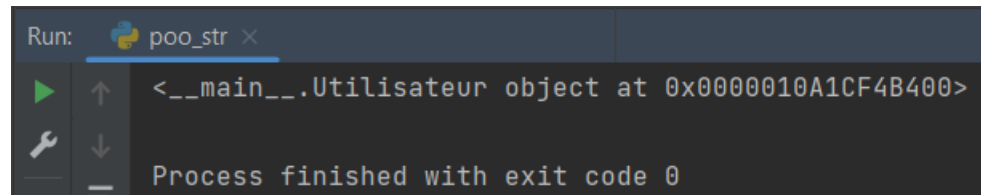
- Nous avons vu précédemment que l'affichage d'une instance n'affiche pas les variables de l'instance mais son type avec une adresse mémoire

- **Exemple :**

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

jeff_bezos = Utilisateur("Bezoz", 59)
print(jeff_bezos)
```



```
Run: poo_str x
<__main__.Utilisateur object at 0x00000010A1CF4B400>
Process finished with exit code 0
```

- **Problématique :** comment faire pour afficher d'autres informations ?

Python- Méthode `__str__`

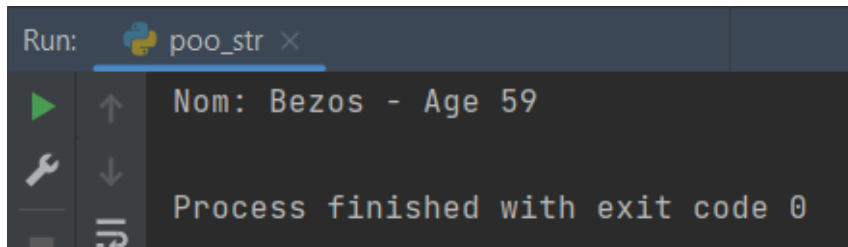
- Pour afficher une chaîne de caractères qui correspond plus à nos besoins, la **solution** est de **redéfinir la méthode `__str__`**
- **Exemple :**

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __str__(self):
        return f"Nom: {self.nom} - Age {self.age}"

jeff_bezos = Utilisateur("Bezos", 59)
print(jeff_bezos)
```



```
Run: poo_str x
Nom: Bezos - Age 59
Process finished with exit code 0
```


Python- Méthode `__repr__`

- Il existe une seconde méthode qu'on peut redéfinir, **la méthode `__repr__`** mais elle ne s'utilise pas tout à fait pour les mêmes besoins.
- **Exemple :**

```
class Utilisateur:

    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def __repr__(self):
        return f"Utilisateur(nom='{self.nom}', prenom='{self.prenom}')
```

Run:  poo_repr x 1

```
Utilisateur(nom='Bezos', prenom='59')
```

Process finished with exit code 0

Python- `__str__` vs `__repr__`

- **Quelle est la différence entre les 2 méthodes ?**
- Les 2 méthodes **sérialisent** un objet, i.e. transforment un objet en chaîne de caractères.
- La bonne pratique pour `__repr__` est de fournir une sérialisation technique permettant la recreation de l'objet avec la fonction **`eval(...)`**
- La bonne pratique pour `__str__` est de fournir une sérialisation fonctionnelle.

Python- `__repr__` avec `eval`

- Afin de pouvoir reconstruire un objet à partir d'une chaîne de caractère en utilisant **`eval`** il est nécessaire de respecter une syntaxe particulière.
- Structure de la chaîne :

```
Classe {var1='valeur1', var2=valeur2}
```

Python- __repr__ avec eval - exemple

➤ Exemple :

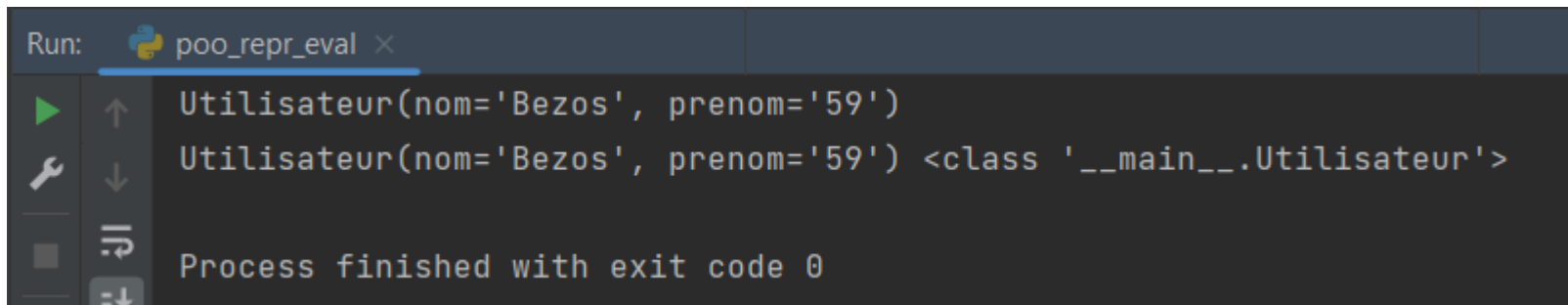
```
class Utilisateur:

    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def __repr__(self):
        return f"Utilisateur(nom='{self.nom}', prenom='{self.prenom}')"

jeff_bezos = Utilisateur("Bezos", 59)
print(repr(jeff_bezos))

jeff_bezos2 = eval(repr(jeff_bezos)):
print(jeff_bezos2, type(jeff_bezos2))
```



```
Run: poo_repr_eval x
Utilisateur(nom='Bezos', prenom='59')
Utilisateur(nom='Bezos', prenom='59') <class '__main__.Utilisateur'>
Process finished with exit code 0
```

Python- méthodes repr() et str()

- La fonction **repr()** permet de convertir une instance en chaîne de caractères en invoquant **__repr__**
- La fonction **str()** permet de convertir une instance en chaîne de caractères en invoquant **__str__**
- **Exemple:**

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __repr__(self):
        return f"Utilisateur(nom='{self.nom}', age='{self.age}')"

    def __str__(self):
        return f"Nom: {self.nom} - Age {self.age}"

jeff_bezos = Utilisateur("Bezos", 59)
print(str(jeff_bezos))
print(repr(jeff_bezos))
```


Python- cas des listes

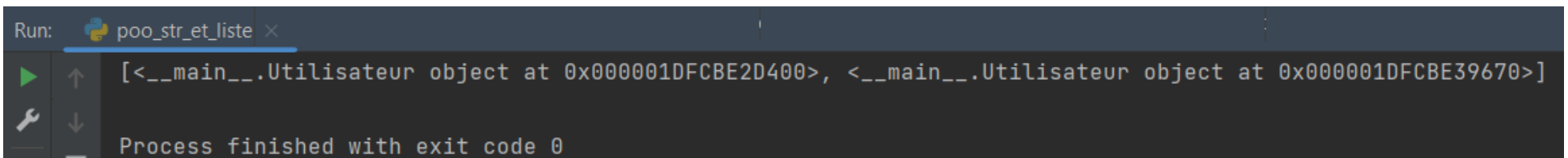
- L'affichage d'une **liste** avec **print()** utilise exclusivement la méthode **__repr__**.
- Dans l'exemple ci-dessous, la méthode **__str__** n'est pas utilisée
- **Exemple:**

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __str__(self):
        return f"Nom: {self.nom} - Age {self.age}"

liste = [Utilisateur("Bezos", 59), Utilisateur("Musk", 52)]
print(liste)
```



```
Run: poo_str_et_liste x
[<__main__.Utilisateur object at 0x000001DFCBE2D400>, <__main__.Utilisateur object at 0x000001DFCBE39670>]
Process finished with exit code 0
```

Atelier (TP)

OBJECTIFS : Savoir redéfinir `__str__` et `__repr__`

DESCRIPTION :

- Dans le TP n°10 vous allez redéfinir les méthodes `__str__` et `__repr__` d'une classe afin d'afficher des infos intelligibles lorsque la fonction `print` affiche des instances de cette classe.



Egalité de 2
instances

Python- Egalité - Problématique

- **Problématique** : étant donné une classe Utilisateur ayant 2 attributs d'instance nom et age, est-ce que l'opérateur d'égalité == fonctionne avec des instances de cette classe ?
- **Faisons le test** :

```
class Utilisateur:  
  
    def __init__(self, nom, age):  
        self.nom = nom  
        self.age = age  
  
jeff_bezos1 = Utilisateur("Bezos", 59)  
jeff_bezos2 = Utilisateur("Bezos", 59)  
  
print(_jeff_bezos1 == jeff_bezos2_)
```

```
Run: poo_egalite_problematique x  
False  
Process finished with exit code 0
```

- **Résultat**: bien que les instances aient les mêmes valeurs d'attributs, le test d'égalité retourne False.

Python- Egalité - Problématique

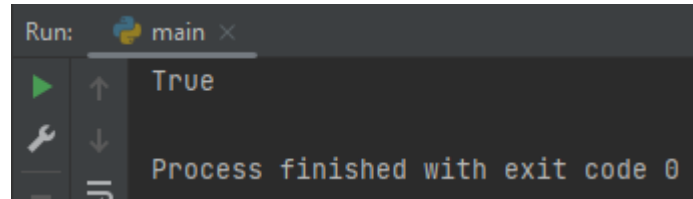
➤ Faisons un second test :

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

jeff_bezos1 = Utilisateur("Bezos", 59)
jeff_bezos2 = jeff_bezos1

print(jeff_bezos1 == jeff_bezos2)
```



Run: main x

True

Process finished with exit code 0

- **Résultat:** Pour que le test d'égalité retourne True, il faut que la seconde instance soit en réalité la même instance que la première.
- jeff_bezos1 et jeff_bezos2 référencent le même objet en mémoire
- 1 seul objet construit mais 2 références
- **Résultat souhaité :** que l'opérateur == compare les valeurs des attributs d'instance et non les adresses mémoire

Python- Egalité - Solution

- Mettre en place la méthode `__eq__(self, other)` :

```
class Utilisateur:

    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def __eq__(self, autre):
        if not isinstance(autre, Utilisateur):
            return False
        return self.nom==autre.nom and self.age==autre.age
```

- Cette méthode permet de tester l'égalité de l'objet courant (**self**) avec un second objet passé en paramètre (**autre**)
- **Etape 1:** si **autre** n'est pas une instance d'**Utilisateur** on renvoie **False**.
- **Etape 2:** on retourne le résultat de la comparaison des attributs des 2 instances.

Atelier (TP)

OBJECTIFS : Savoir redéfinir `__eq__`

DESCRIPTION :

- Dans le TP n°11 vous allez redéfinir la méthode `__eq__` d'une classe afin de pouvoir comparer 2 instances d'une classe avec l'opérateur `==`.



Méthode __hash__ pour les sets

Python- Egalité – Problème avec les sets et les dicts

- Une fois la méthode `__eq__` redéfinie, vos instances ne sont plus stockables dans un set ou dans un dictionnaire :

```
class Utilisateur:

    def __init__(self, nom, age):...

    def __eq__(self, autre):...

ut1 = Utilisateur("Bezos", 59)
ut2 = Utilisateur("Musk", 50)

sets = {ut1, ut2}
print(sets)
```

```
Run: poo_egalite_solution x
Traceback (most recent call last):
  File "C:\Users\RichardBONNAMY\PycharmProjects\preparation
    sets = { ut1, ut2}
TypeError: unhashable type: 'Utilisateur'
```



- A l'exécution vous obtenez le message suivant : **unhashable** type Utilisateur

Python- Egalité – Explication du problème

- Le stockage d'un objet dans un set utilise le code de hashage de l'objet.
- Le set va appeler la méthode `__hash__` de votre classe pour calculer ce code et trouver un emplacement pour l'objet.
- Cette méthode fonctionne avec la méthode `__eq__` selon le principe suivant:
 - **Si `a == b` alors `hash(a) == hash(b)`**
- Dans le cas où **vous avez redéfini la méthode `__eq__` mais pas la méthode `__hash__` vous avez brisé ce principe.**
- Votre objet n'est plus hashable.

Python- Egalité – Solution du problème

- Il faut **redéfinir** la méthode **__hash__**
- Comment faire ? Vous pouvez utiliser la fonction native **hash()** en lui passant en paramètre l'ensemble de vos attributs encapsulés dans un tuple.
- **Exemple pour Utilisateur :**

```
class Utilisateur:

    def __init__(self, nom, age):...

    def __eq__(self, autre):...

    def __hash__(self):
        return hash((self.nom, self.age))
```

```
Run: poo_egalite_solution x
{<__main__.Utilisateur object at 0x0000002BB3F98C5B0>, <__main__.Utilisateur object at 0x0000002BB3F29B400>}
Process finished with exit code 0
```

Atelier (TP)

OBJECTIFS : Savoir redéfinir `__eq__`

DESCRIPTION :

- Dans le TP n°12 vous allez redéfinir la méthode `__hash__` d'une classe afin de pouvoir stocker des instances de cette classe dans un set.

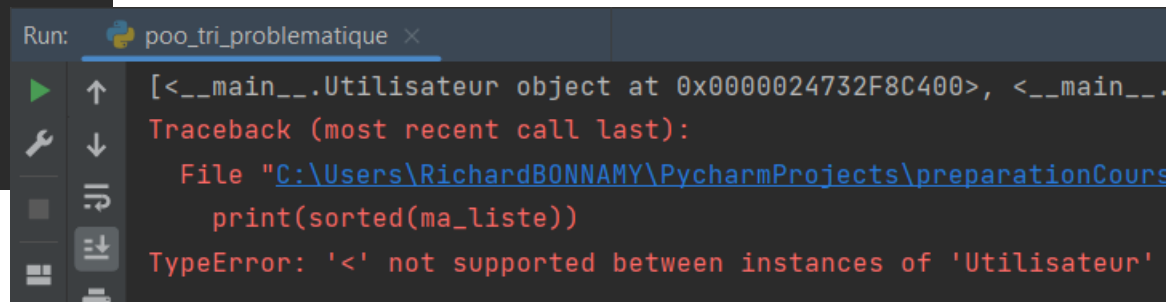


Trier vos listes

Python- Tri - Problématique

- **Problématique** : étant donné une classe Utilisateur ayant 2 attributs d'instance nom et age, est-ce que le tri d'une liste d'utilisateurs fonctionne ?
- **Faisons le test :**

```
class Utilisateur:  
  
    def __init__(self, nom, age):  
        self.nom = nom  
        self.age = age  
  
ut1 = Utilisateur("Dupont", 34)  
ut2 = Utilisateur("Lee", 22)  
ut3 = Utilisateur("Bakti", 30)  
ut4 = Utilisateur("Johnson", 42)  
  
ma_liste = [ut1, ut2, ut3, ut4]  
print(ma_liste)  
print(sorted(ma_liste))
```



```
Run: poo_tri_problematique x  
[<__main__.Utilisateur object at 0x0000024732F8C400>, <__main__...  
Traceback (most recent call last):  
  File "C:\Users\RichardBONNAMY\PycharmProjects\preparationCours...  
    print(sorted(ma_liste))  
TypeError: '<' not supported between instances of 'Utilisateur'
```

- A l'exécution vous obtenez le message suivant : '<' not supported between instances of 'Utilisateur'

Python- Tri – Explication du problème

- La **fonction sorted** ne sait pas comment trier vos instances de classe
- Rien dans votre classe n'explique comment faire le tri

Python- Tri – Solution du problème

- Il faut **redéfinir** la méthode `__lt__` qui permet de comparer l'instance courante (**self**) à une autre passée en paramètre (**autre**)
- **Exemple pour Utilisateur :**

```
class Utilisateur:  
  
    def __init__(self, nom, age):...  
  
    def __repr__(self):...  
  
    def __lt__(self, autre):  
        return self.age < autre.age
```

```
ut1 = Utilisateur("Dupont", 34)  
ut2 = Utilisateur("Lee", 22)  
ut3 = Utilisateur("Bakti", 30)  
ut4 = Utilisateur("Johnson", 42)  
  
ma_liste = [ut1, ut2, ut3, ut4]  
print(ma_liste)  
print(sorted(ma_liste))
```

```
Run: poo_tri_solution x  
[Dupont - 34 , Lee - 22 , Bakti - 30 , Johnson - 42 ]  
[Lee - 22 , Bakti - 30 , Dupont - 34 , Johnson - 42 ]
```


Python- Tri – Méthode `__cmp__` vs `__lt__`

- La méthode `__cmp__` était utilisée en **Python 2**
- Elle n'est plus supportée en **Python 3** qui utilise `__lt__`

Python- Opérateur de comparaison <

- Le fait de **redéfinir** la méthode **`__lt__`** redéfinit aussi l'opérateur de **comparaison <** pour votre classe
- **Exemple pour Utilisateur :**

```
class Utilisateur:

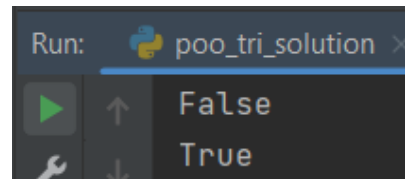
    def __init__(self, nom, age):...

    def __repr__(self):
        return f"{self.nom} - {self.age} "

    def __lt__(self, autre):
        return self.age < autre.age

ut1 = Utilisateur("Dupont", 34)
ut2 = Utilisateur("Lee", 22)
ut3 = Utilisateur("Bakti", 30)

print( ut1 < ut2 )
print( ut2 < ut3 )
```



Python- Tri externalisé

- Si on veut externaliser le critère de tri, il faut passer en paramètre de la méthode **sorted** une fonction qui prend en paramètre un utilisateur et retourne un critère de tri.
- **Exemple pour Utilisateur :**

```
def tri_service(salarie):  
    return salaire.service  
  
def tri_service_nom(salarie):  
    return (salarie.service, salarie.nom)  
  
liste_triee = sorted(lst1, key=tri_service_nom)  
print(liste_triee)
```

- Si on utilise **key=tri_service** on tri sur le service
- Si on utilise **key=tri_service_nom** on tri sur le service, puis sur le nom

Atelier (TP)

OBJECTIFS : Savoir redéfinir `__lt__`

DESCRIPTION :

- Dans le TP n°13 vous allez redéfinir la méthode `__lt__` d'une classe afin de pouvoir trier une liste d'instance de cette classe



Gérer les exceptions

Python- Création d'une exception

- Pour une exception, il faut créer une classe qui hérite de la classe Exception ou d'une de ses sous-classes.
- Exemple :

```
class MonExceptionPersonnalisee(Exception):  
    def __init__(self, message):  
        super().__init__(message)
```

Python- Levée d'une exception

- Pour lever (ou jeter) une exception, il faut utiliser le mot clé **raise**.
- **Exemple :**

```
class Service:

    def insererDonnees(self, utilisateur):
        if not utilisateur.nom:
            raise MonExceptionPersonnalisee("Le nom est obligatoire")
        if not utilisateur.age:
            raise MonExceptionPersonnalisee("L'age' est obligatoire")
```

- Dans l'exemple ci-dessus on jette des exceptions dans le cas où l'utilisateur passé en paramètre n'a pas les informations attendues.

Python- Traiter une exception

- Pour traiter une exception il faut utiliser **try / except**
- **Exemple :**

```
service = Service()
try:
    service.insererDonnees(Utilisateur(59, ""))
except MonExceptionPersonnalisee as e:
    print(e)
```

- On "essaye" d'exécuter la méthode à risque dans le bloc **try**.
- Si une exception de type **MonExceptionPersonnalis  e** se produit, la clause **except** traite cette derni  re et affiche le message d'erreur.

Python- Traiter plusieurs types d'exception

- Il est possible d'utiliser un seul except pour traiter plusieurs types d'exception.
- **Exemple :**

```
service = Service()
try:
    service.insererDonnees(Utilisateur(59, ""))
except (MonExceptionPersonnalisee, ValueError) as e:
    print(e)
```

- Dans ce cas on met la liste des types entre parenthèses séparés par des virgules.

Python- Plusieurs blocs except

- Il est possible d'utiliser plusieurs blocs **except**.
- **Exemple :**

```
service = Service()
try:
    service.insererDonnees(Utilisateur(59, ""))
except MonExceptionPersonnalisee as e:
    print(e)
except Exception as e:
    print("une exception inattendue s'est produite :" + e)
```

- **Attention** : si vous inversez les 2 blocs, et que vous traitez d'abord **except Exception**, le bloc suivant ne sera jamais exécuté. Pas d'erreur à l'exécution mais il ne sera pas exécuté.

Python- except / else

- Il est possible d'exécuter un bloc de code dans le cas particulier où l'exception ne se produit pas.

- **Exemple :**

```
service = Service()
try:
    service.insererDonnees(Utilisateur(59, "Bezos"))
except MonExceptionPersonnalisee as e:
    print(e)
else:
    print("Tout s'est bien déroulé !")
```

- Mais on peut utiliser cette écriture :

```
service = Service()
try:
    service.insererDonnees(Utilisateur(59, "Bezos"))
    print("Tout s'est bien déroulé !")
except MonExceptionPersonnalisee as e:
    print(e)
```

Python- intérêt du except / else

- L'intérêt se situe plus au niveau de la structuration du code et de sa lisibilité pour séparer par exemple des traitements.
- **Exemple :**

```
service = Service()
try:
    service.insererDonnees(Utilisateur(59, "Bezos"))
except MonExceptionPersonnalisee as e:
    notifierErreur()
else:
    notifierSucces()
```

Python- except / else / finally

- Il est possible d'exécuter un bloc de code dans tous les cas, que cela se passe bien ou non.
- On utilise pour cela le bloc **finally**
- **Exemple :**

```
service = Service()
try:
    service.insererDonnees(Utilisateur(59, "Bezos"))
except MonExceptionPersonnalisee as e:
    notifierErreur()
else:
    notifierSucces()
finally:
    fermerConnexion()
```

Atelier (TP)

OBJECTIFS : Savoir utiliser les exceptions

DESCRIPTION :

- Dans le TP n°14 vous allez mettre en place une classe de service de type validation qui contrôle qu'un utilisateur a des données correctes.



Documentation

Python- Documenter ses classes

- La documentation d'une classe est un aspect critique du développement professionnel.
- Vous êtes avant tout un auteur, qui écrit pour être lu.
- Outre la lisibilité de votre code, principe **KISS**: Keep It Simple and Stupid, **vous devez documenter votre code.**



Python- Les docstrings

- La meilleure façon de documenter ses classes et ses méthodes est d'utiliser les **docstrings**

"""Je suis la documentation"""

- **Exemple :**

Python

```
class SimpleClass:
    """Class docstrings go here."""

    def say_hello(self, name: str):
        """Class method docstrings go here."""

        print(f'Hello {name}')
```

Python- Les docstrings

- La meilleure façon de documenter ses classes et ses méthodes est d'utiliser les **docstrings**

"""Je suis la documentation"""

- Vous devez documenter la classe et chacune des méthodes
- Pour chaque méthode vous devez documenter les paramètres et le return
- **Exemple :**

```
class Operation:
    """
    Classe utilitaire proposant des opérations mathématiques
    """

    def addition(self, a: int, b: int) -> int:
        """
        Cette méthode réalise une addition
        :param a: entier 1
        :param b: entier 2
        :return: un entier résultat de l'addition
        """
        return a + b
```

Python- Aide sous PyCharm

- Positionner vous sous la définition de la méthode
- Ouvrez votre docstring avec `"""` puis appuyez sur **Enter**
- PyCharm génère un template de docstring avec les paramètres et le type de retour
- **Exemple :**

```
class Operation:
    """
    Classe utilitaire proposant des opérations mathématiques
    """

    def addition(self, a: int, b: int) -> int:
        """
        :param a:
        :param b:
        :return:
        """
        return a + b
```

Python- Fonction help(nom_classe)

➤ Vous pouvez afficher la documentation d'une classe avec la fonction help

➤ **Exemple :**

```
class Operation:
    """
    Classe utilitaire proposant des opérations mathématiques
    """

    def addition(self, a: int, b: int) -> int:
        """
        Cette méthode réalise une addition
        :param a: entier 1
        :param b: entier 2
        :return: un entier résultat de l'addition
        """
        return a + b
```

```
help(Operation)
```

```
Help on class Operation in module __main__:

class Operation(builtins.object)
|  Classe utilitaire proposant des opérations mathématiques
|
|  Methods defined here:
|
|  addition(self, a: int, b: int) -> int
|      Cette méthode réalise une addition
|      :param a: entier 1
|      :param b: entier 2
|      :return: un entier résultat de l'addition
|
|  -----
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

Process finished with exit code 0

FIN

MERCI DE VOTRE ATTENTION !