Lors de cette étude, nous allons enfin abordé l'un des paradigmes les plus importants de ces dernières années dans le monde du développement, la programmation objet. Python l'utilise très largement puisque tous les types composés sur lesquels nous avons déjà travaillé (chaînes, listes, tuples, dictionnaires, ensembles, etc.) sont des objets.

Dès que vous avez besoin d'une structure un peu complexe, c'est beaucoup plus facile de passer par ce style de programmation. L'avantage également, c'est que vous allez pouvoir créer vos propres types.

Quelques notions sur les objets

Les objets sont des entités (des variables) qui possèdent intérieurement une ou plusieurs autres variables, que nous nommons **« attributs »**, mais plus surprenant encore, qui intègrent également des fonctionnalités associées à ces attributs, que nous nommons **« méthodes »** qui décrivent tous les comportements possibles relatifs à ces objets.

*La programmation objet est extrêmement séduisante puisque nous n'avons plus les **« données »** d'un côté et les **« fonctions »** de l'autre, mais au contraire une fusion **(encapsulation)** entre ces deux éléments. Ainsi, les **« méthodes »** sont parfaitement adaptées à ce que peut faire réellement l'objet en question. Les **« attributs »** ne sont pas accessibles directement, seules les **« méthodes »** le sont, ce qui se fait très simplement au travers de l'opérateur point **« . »**.*

*La description de l'ensemble de ces éléments **(« attributs » + « méthodes »)** est déclarée dans ce que nous appelons une **« classe »**. En réalité, une **« classe »** est tout simplement un **« type »**.*

***Objet = identité** (nom de la variable) **+ état** (valeurs des attributs) **+ comportement** (méthodes associées).*

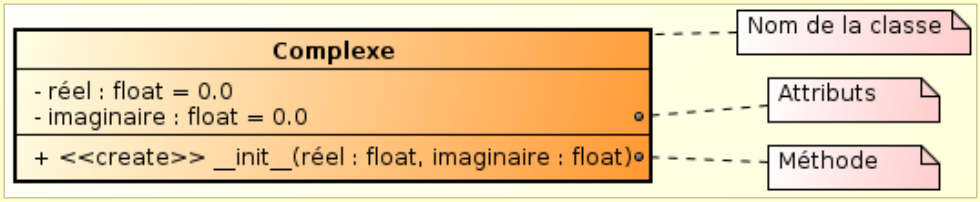
Comme exemple, prenons un nombre complexe. Il n'est pas implémenté nativement dans Python (il existe toutefois la classe « complex »** dans le module **« math »). Il possède une partie réelle et une partie imaginaire dont les valeurs sont indépendantes, et en même temps cela forme un tout indissociable. Par ailleurs, nous pouvons effectuer un certain nombre de traitements spécifiques, comme le calcul du module ou de l'argument, ou encore des opérations arithmétiques, comme l'addition, la soustraction, la multiplication, etc. C'est vraiment l'exemple typique où il est judicieux de proposer une nouvelle classe (type) qui possèdent toutes ces particularités là. Tout est inscrit à l'intérieur de cette classe avec sa propre autonomie.

Création de nouvelles classes avec leurs attributs et création de nouveaux objets

Je vous propose dans une premier temps de créer deux nouvelles classes (deux nouveaux types), l'une **« Complexe »** représentant les nombres complexes, l'autre **« Personne »** représentant l'identité d'une personne.

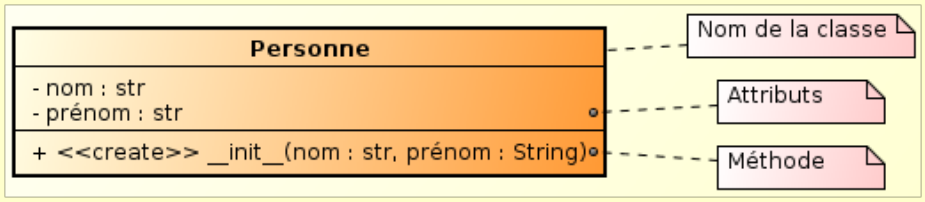
Remarquez au passage que lorsque nous créons de nouveaux types, même si cela n'est pas obligatoire, il est préférable que la première lettre du nom de la classe soit en lettre majuscule (cela permet de les différencier des types déjà existants).

complexe.py




```
class Complexe:

   # définition du constructeur (méthode utilisée durant la phase de création)
   def __init__(self, réel=0.0, imaginaire=0.0):
      self.réel = réel                  # attribut 'réel'
      self.imaginaire = imaginaire      # attribut 'imaginaire'
```

personne.py




```
class Personne:

   # définition du constructeur (méthode utilisée durant la phase de création)
   def __init__(self, nom, prénom):
      self.nom = nom.upper()              # attribut 'nom'
      self.prénom = prénom.title()        # attribut 'prénom'
```

principal.py

```
# importation des classes
from complexe import Complexe
from personne import Personne

# Création de plusieurs objets a, b, c de la classe 'Complexe'
a = Complexe(5.3, 8.9)
b = Complexe(2.5)
```

```
c = Complexe()
c.imaginaire = -1

# Création d'un objet de la classe 'Personne'
moi = Personne('rémy', 'emmanuel')

# Affichage du contenu des différents objets
motif = '{} : (réel={}, imaginaire={})'
print(motif.format('a', a.réel, a.imaginaire))
print(motif.format('b', b.réel, b.imaginaire))
print(motif.format('c', c.réel, c.imaginaire))

print(moi.prénom, moi.nom)
```

Shell Python : [évaluer principal.py]

```
a : (réel=5.3, imaginaire=8.9)
b : (réel=2.5, imaginaire=0.0)
c : (réel=0.0, imaginaire=-1)
Emmanuel RÉMY
```

Créer une nouvelle classe

Une **« classe »** représente une structure complète de notre nouveau type, en fusionnant toutes les données internes utiles et indispensables pour ce nouveau type, qui s'appellent donc les **« attributs »** avec l'ensemble également des fonctions internes, les **« méthodes »** qui vont décrire le fonctionnement attendu en rapport avec ce nouveau type.

Une « classe » est donc un moule avec lequel nous pourrons construire toutes les pièces nécessaires, c'est-à-dire les « objets ». Nous avons exactement la même chose lorsque nous créons des variables (pièces) par rapport à un type particulier (le moule) comme pour le type entier par exemple.

*Pour créer une nouvelle **« classe »**, nous utilisons le mot réservé **« class »** suivi de son nom unique avec la première lettre en majuscule, suivi, comme d'habitude du **« : »**. Tout ce qui devra être intégré à la classe, **« attributs »** et **« méthodes »** devront être indentés pour bien délimiter le bloc d'instructions, comme nous l'avons déjà fait pour la définition des fonctions.*

Créer les attributs associés à cette classe en passant par le constructeur

Pour définir les attributs, Python ne s'y prend pas comme les autres langages. Vous devez passer systématiquement par une méthode spéciale qui s'appelle un constructeur, nommé ici **« __init__ »**. Toutes les méthodes spéciales dans Python utilisent la même syntaxe par le double souligné **« __ »** de part et d'autre du nom de la méthode.

Un constructeur est une méthode dite de rappel, qui est automatiquement sollicitée lorsque nous créons un nouvel objet. Je le rappelle, une méthode est tout simplement une fonction intégrée dans une classe.

*La façon de la définir est exactement la même que lors de la définition d'une fonction classique, comme nous l'avons fait lors de l'étude précédente. Ainsi, nous commençons par le mot réservé **« def »** suivi du nom de la méthode, suivi de la liste des paramètres entre parenthèses **« () »**, chacun séparés par des virgules **« , »**. Vous clôturez enfin vos définitions par l'opérateur **« : »**. Comme pour les fonctions, il est également possible de proposer des valeurs par défaut aux paramètres.*

*Pour nos deux classes spécifiques, chacun des constructeurs propose deux paramètres précédés systématiquement par un paramètre supplémentaire, **« self »**. Ce paramètre est fondamental puisqu'il correspond à l'objet qui est en train d'être créé. Par exemple, pour la classe **« Complexe »**, **« self »** représente respectivement les objets **« a »**, **« b »** et **« c »** que nous découvrons dans le script **« principal.py »**.*

*À partir de l'objet créé, nous pouvons utiliser les **« attributs »** et les **« méthodes »** intégrés, en précisant d'abord le nom de l'objet concerné, suivi de l'opérateur de séparation **« . »** et suivi ensuite de vos **« attributs »** ou de vos **« méthodes »**, comme nous l'avons déjà expérimenté, lorsque nous avons par exemple, utilisé la méthode **« append() »** d'une liste.*

Pour en revenir à la création de nos « attributs », il suffit donc de les désigner dans le constructeur à l'aide du paramètre « self ». Chaque élément intégré grâce à « self » fait alors définitivement parti de « l'objet ». Dans ce cadre là, tous les « objets » auront donc exactement les mêmes noms d'attributs (décrits dans la « classe »), par contre chaque « objet » proposera ses propres valeurs et sera donc dans un état particulier à ce moment là, état qui peut bien entendu évoluer par la suite.

Objet = identité** (nom de la variable) **+ état** (valeurs des attributs) **+ comportement** (méthodes associées)**.

*Le constructeur, mis à part son exécution automatique, est une **« méthode »** comme une autre, et à ce titre, nous pouvons prévoir un certain nombre d'actions bien utiles pour renseigner correctement les nouveaux **« attributs »**. C'est ce que nous faisons dans la classe **« Personne »** où lors de la récupération de l'identité de la personne, nous proposons de mettre le nom tout en majuscule et le prénom avec uniquement la première lettre en majuscule.*

Création et utilisation des objets

Lorsque nous créons nos objets, vous le faites en spécifiant le nom de la classe correspondante avec des parenthèses, avec éventuellement les arguments requis, si les constructeurs possèdent des paramètres. Je rappelle que **« self »** ne fait pas réellement parti des paramètres attendus puisqu'il représente directement l'objet à gauche de l'opérateur **« = »**.

Une fois qu'un objet est créé, il peut évoluer dans le temps. Il est possible qu'il change d'état, notamment lorsque vous faites appel à une méthode spécifique de l'objet, où ici lorsque nous changeons, en cours de route la valeur d'un attribut.

*C'est justement ce qui est fait pour l'objet **« c »**, lorsque nous lui proposons une nouvelle valeur pour l'attribut **« imaginaire »** :*

```
c.imaginaire = -1
```

Attributs publics ou privés

Par défaut dans Python, les attributs sont **« publics »**, c'est-à-dire qu'il est possible de les modifier directement à l'aide de l'objet créé comme nous venons de le faire dans la ligne de code précédente. Toutefois, normalement dans la philosophie objet, ils doivent impérativement être privés. C'est d'ailleurs ce que montrent les diagrammes des classes, dans la zone d'attributs, vous remarquez la présence du signe moins devant chaque attribut, ce qui veut dire **« privé »**.

*Dans le cas de la classe **« Complexe »**, nous pouvons conserver le statut de **« public »** pour ces attributs là, cela n'a aucune conséquence néfaste. Par contre, pour la classe **« Personne »**, c'est plus embêtant. Si par exemple, nous décidons de changer le **« prénom »** à posteriori**,** il serait plus agréable de conserver le système automatique qui permet de mettre la première lettre en majuscule.*

Hors, par défaut, vous êtes libre de changer votre prénom, mais ce système automatique n'est bien entendu pas pris en compte, puisqu'il s'agit d'un comportement spécifique qui doit donc être décrit dans une méthode particulière (ce n'est pas magique, tout traitement doit bien être écrit quelque part). Voici d'ailleurs un exemple de changement :

principal.py

```
from personne import Personne

moi = Personne('rémy', 'emmanuel')
print(moi.prénom, moi.nom)

moi.prénom = 'alain'
print(moi.prénom, moi.nom)
```

Shell Python : [évaluer principal.py]

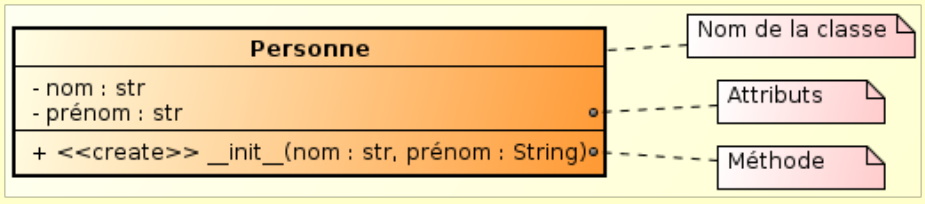
```
Emmanuel RÉMY
alain RÉMY # le prénom conserve la chaîne proposé directement sur l'attribut sans mise en majuscule
```

```
1  from personne import Personne
2
3  moi = Personne('rémy', 'emmanuel')
4  print(moi.prénom, moi.nom)
5  moi.prénom = 'alain'
6                    , moi.nom)
       nom
       prénom
       __class__
       __init__
```

Remarquez ci-contre, lorsque nous utilisons la complétion, l'ensemble des éléments constituant l'objet, notamment les attributs ainsi que le constructeur, apparaissent automatiquement :

*Vous pouvez décider, et c'est ce qu'il est préférable de faire dans la majorité des cas, de rendre **« privé »** vos attributs. Ici, cela pourrait être judicieux afin d'empêcher le changement d'identité ultérieurement. La solution est simple, il suffit de préfixer vos noms d'attributs du double souligné **« __ »**, qui je le rappelle a une connotation particulière.*

personne.py




```
class Personne:

   # définition du constructeur (méthode utilisée durant la phase de création)
   def __init__(self, nom, prénom):
      self.__nom = nom.upper()              # attribut '__nom'
      self.__prénom = prénom.title()        # attribut '__prénom'
```

```
1  from personne import Personne
2
3  moi = Personne('rémy', 'emmanuel')
4  print(moi.__prénom, moi.__nom)
5  moi.__
6  print(
       __class__
       __init__
                  i.__nom)
```

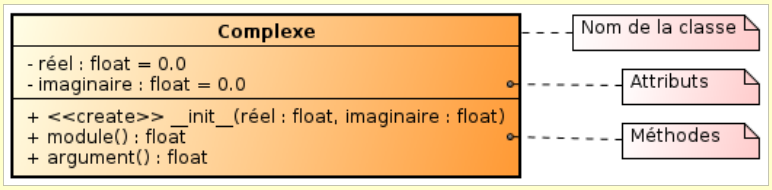
*Effectivement, maintenant nous ne voyons plus du tout les attributs. Ils sont automatiquement cachés (**« privés »**). Par contre, nous ne pouvons plus les utiliser, même pour afficher leurs valeurs, puisqu'ils ne sont plus du tout accessibles (**« privés »**). Nous verrons bientôt comment résoudre ce petit problème.*

Ajout de nouvelles méthodes à la classe « Complexe »

Je vous propose de rajouter deux nouvelles méthodes à la classe **« Complexe »** qui nous permettra de connaître à tout moment le module et l'argument d'un nombre complexe particulier.

complexe.py

```
from math import *

class Complexe:
   def __init__(self, réel=0.0, imaginaire=0.0):
      self.réel = réel                  # attribut 'réel'
      self.imaginaire = imaginaire      # attribut 'imaginaire'

   def module(self): return sqrt(self.réel**2 + self.imaginaire**2)

   def argument(self):
      if self.imaginaire == 0: return 0
      if self.réel == 0:
         if self.imaginaire >= 0: return pi/2
         else: return -pi/2
      return atan(self.imaginaire/self.réel)
```



principal.py

```
from complexe import Complexe

z = Complexe(1, -1)
print('module =', z.module())
print('argument =', z.argument())
```

Shell Python : [évaluer principal.py]

```
module = 1.4142135623730951
argument = -0.7853981633974483
```

Dans une classe, nous pouvons créer autant de méthodes que vous désirez. Bien entendu, elles n'ont de sens que par rapport à la classe que vous implémentez. Ainsi, la notion de module et d'argument est bien associée à la notion de nombre complexe.

Vous remarquez que ces méthodes nous donnent un renseignement par rapport à l'état de l'objet actuel. Vous n'avez pas besoin de fournir de valeurs particulières au moment de l'appel de ces méthodes, puisque tout ce que nous avons besoin est déjà intégré dans l'objet grâce à ses attributs (l'objet garde constamment en mémoire la valeur de tous ses attributs).

C'est une des raisons qui fait que la programmation objet est vraiment géniale.

*Tout ce que vous avez besoin pour réaliser vos calculs, se trouve déjà dans l'objet, avec ses attributs **« réel »** et **« imaginaire »**. C'est pour cette raison que nous avons systématiquement rajouté le paramètre implicite **« self »** (objet qui appelle actuellement la méthode) à chacune de nos méthodes. Nous en avons besoin pour récupérer la valeur des attributs associés à l'objet en cours.*

*Comme pour les attributs, lorsque vous voulez utiliser une méthode particulière, vous devez d'abord choisir l'objet sur lequel porte le calcul (ils seront différents si les objets ont des états différents), utiliser ensuite l'opérateur de séparation **« . »** et enfin designer la méthode. N'oubliez pas les parenthèses **« () »**, même si vous n'avez pas à fournir d'arguments comme c'est le cas pour ces deux méthodes.*

Les propriétés

Le principe des propriétés est intimement lié à la notion d'encapsulation qui permet deux choses fondamentales, d'une part, la fusion entre les données et les traitements associés (**« attributs »** + **« méthodes »**), et d'autre part, le fait que les attributs ne soient pas accessibles directement à l'extérieur de l'objet lui-même, attributs **« privés »**.

*L'idée générale, c'est d'éviter de mettre n'importe quelle valeur non adaptée à un attribut. Nous devons systématiquement passer par une méthode spécifique qui permet de contrôler la situation. De toute façon, par principe, l'objet change très souvent d'état (**« attributs »** modifiés) à l'issue d'une ou plusieurs actions (**« méthodes »**) particulières.*

Si par exemple, vous voulez changer la couleur de la voiture (changer d'état), ce n'est pas magique, vous êtes bien obligé de passer par tout un tas de procédures particulières : décaper, poncer, passer la nouvelle peinture, vernir, etc. C'est à l'issue de toutes ces opérations que le changement de couleur est opéré.

Les propriétés sont un moyen transparent de manipuler indirectement les attributs d'un objet. Elles permettent de dire à Python : « Quand un utilisateur souhaite modifier cet attribut, il passera automatiquement par une série d'actions adaptées, sans que l'utilisateur en ait conscience ».

De cette façon, nous pouvons rendre certains attributs totalement inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture seule, ou encore, nous pouvons faire en sorte que, si nous modifions effectivement un attribut, la méthode recalcule alors la valeur d'autres attributs de l'objet (le changement d'un attribut a des conséquences sur les autres attributs).

Une propriété représente un attribut, et possède au maximum trois éléments distincts :

- ***L'attribut lui-même*** *(optionnel) qui doit être privé,*
- ***Une méthode en lecture seule*** *(obligatoire), dite* ***« accesseur »*** *qui permet de retourner la valeur de l'attribut,*
- ***Une méthode en écriture*** *(optionnel), dite* ***« mutateur »*** *qui permet de modifier l'attribut ou d'autres attributs si nécessaire.*

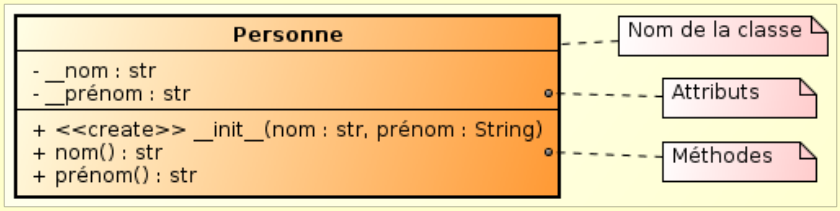
Propriété en lecture seule

Nous avons passé pas mal de temps à expliquer les différents concepts. C'était nécessaire pour bien comprendre les principes, mais je vous rassure la mise en œuvre est vraiment très simple.

Vu que nous prévoyons une propriété en lecture seule, il faut juste créer la méthode ***« accesseur »****. Le choix du nom est très important puisque c'est le nom qui sera visible à l'utilisation. Pensez bien que nous devons avoir un attribut* ***« privé »****, donc précédé de* ***« __ »*** *. Le meilleur choix consiste à choisir le nom de la méthode* ***« accesseur »*** *qui porte le même nom que l'attribut sans ce préfixe.*

L'utilisateur ne voit plus du tout l'attribut réel, puisqu'il est « privé », il voit uniquement la méthode en lecture seule que nous utilisons d'ailleurs comme un attribut, c'est-à-dire sans les parenthèses. Pour indiquer que cette méthode doit être une propriété, Python utilise la technique des « décorateurs », vous devez alors préfixer votre méthode du terme « @property » et le tour est joué.

personne.py




```
class Personne:
   # définition du constructeur (méthode utilisée durant la phase de création)
   def __init__(self, nom, prénom):
      self.__nom = nom.upper()                # attribut '__nom'
      self.__prénom = prénom.title()          # attribut '__prénom'

   @property
   def nom(self) : return self.__nom          # propriété en lecture 'nom'

   @property
   def prénom(self) : return self.__prénom    # propriété en lecture 'prénom'
```

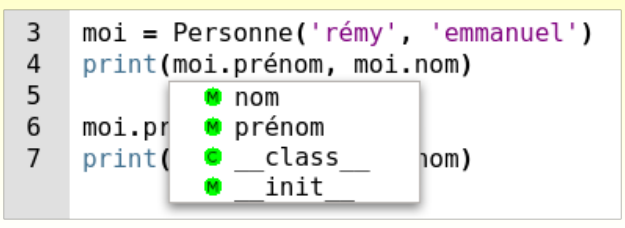
principal.py

```
from personne import Personne

moi = Personne('rémy', 'emmanuel')
print(moi.prénom, moi.nom)

moi.prénom = 'alain'            # INTERDIT : propriété en lecture seule
print(moi.prénom, moi.nom)
```

Shell Python : [évaluer principal.py]

```
Emmanuel RÉMY
```

Dans le code précédent nous utilisons nos méthodes comme s'il s'agissait d'attributs, sans rajouter les parenthèses. Rien n'empêche de les utiliser, mais cela permet d'avoir une lecture plus intuitive et raccourcie. Ce système de propriétés est très agréable, puisqu'il est simple à utiliser et en même temps, nous protégeons les objets qui l'utilise puisque dans ce cas là, nous empêchons le changement d'identité.

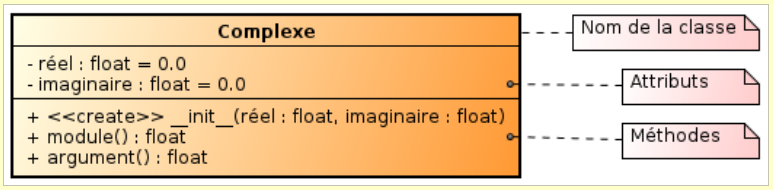


Avec le système de complétion de code, nous ne voyons pas les attributs privés, mais nous avons bien à notre disposition les propriétés associées (qui représentent donc les attributs).

complexe.py

```
from math import *

class Complexe:
   def __init__(self, réel=0.0, imaginaire=0.0):
      self.réel = réel                 # attribut 'réel'
      self.imaginaire = imaginaire     # attribut 'imaginaire'
```

```
   @property
   def module(self): return sqrt(self.réel**2 + self.imaginaire**2)

   @property
   def argument(self):
      if self.imaginaire == 0: return 0
      if self.réel == 0:
         if self.imaginaire >= 0: return pi/2
         else: return -pi/2
      return atan(self.imaginaire/self.réel)
```



principal.py

```
from complexe import Complexe

z = Complexe(1, -1)
print('module =', z.module)          # 'module' est considéré comme un attribut
print('argument =', z.argument)      # 'argument' est considéré comme un attribut
```

Shell Python : [évaluer principal.py]

```
module = 1.4142135623730951
argument = -0.7853981633974483
```

*En revenant sur la classe **« Complexe »**, vu que les deux méthodes sont des méthodes qui renvoient un résultat sans fournir d'argument, elles peuvent être considérées comme des méthodes **« accesseurs »**, même si derrière elles ne sont rattachées à aucun attribut. Du coup, nous pouvons en faire des propriétés en lecture seule.*

Vu de l'extérieur, la classe « Complexe » semble posséder quatre attributs : « réel », « imaginaire », « module » et « argument », ce qui n'est pas illogique, puisque ces attributs font bien partis de la notion de nombre complexe.

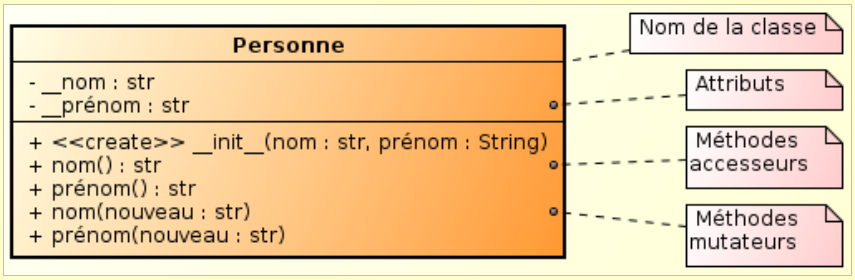
Propriété en lecture et en écriture

Toujours en utilisant ce principe de propriété qui représente l'attribut, nous allons maintenant faire en sorte que nous puissions changer la valeur de l'attribut concerné. Toutefois, nous passons là aussi par une méthode qui va permettre d'ajuster la valeur, comme par exemple, la mise en majuscule automatique du nom de la personne, comme nous l'avons déjà fait dans le constructeur.

*Pour mettre en œuvre une propriété en écriture, nous utilisons de nouveau les décorateurs. Par contre, nous avons également besoin de la propriété en lecture générée précédemment. Là aussi, la mise en œuvre est très simple. Il suffit de respecter la syntaxe suivante : **@nom_propriété_lecture.setter**.*

*Ce décorateur est placé en amont de la méthode, dite **« mutateur »**, qui va servir à récupérer l'argument souhaité et de renseigner ainsi l'attribut correspondant, par un traitement spécifique, afin de respecter le canevas prévu dans le cadre de son utilisation ultérieure.*

personne.py




```
class Personne:
   # définition du constructeur (méthode utilisée durant la phase de création)
   def __init__(self, nom, prénom):
      self.__nom = nom.upper()           # attribut '__nom'
      self.__prénom = prénom.title()     # attribut '__prénom'

   @property
   def nom(self): return self.__nom                # propriété 'nom' en lecture

   @nom.setter
   def nom(self, nouveau): self.__nom = nouveau.upper()    # propriété 'nom' en écriture

   @property
   def prénom(self): return self.__prénom                    # propriété 'prénom' en lecture

   @prénom.setter
   def prénom(self, nouveau): self.__prénom = nouveau.title()  # propriété 'prénom' en écriture
```

principal.py

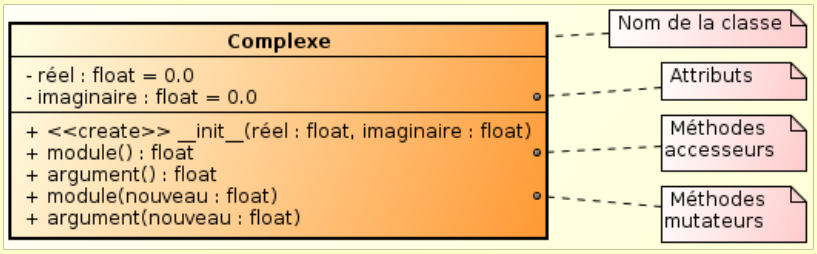
```
from personne import Personne

moi = Personne('rémy', 'emmanuel')
print(moi.prénom, moi.nom)

moi.prénom = 'jean-michel'            # Autorisé cette fois-ci, propriété en écriture
print(moi.prénom, moi.nom)
```

Shell Python : [évaluer principal.py]

```
Emmanuel RÉMY
Jean-Michel RÉMY
```

Encore fois, ce système de propriétés est vraiment bien conçu, puisque l'utilisateur a l'impression d'accéder directement aux attributs de la classe, alors qu'il passe systématiquement par des méthodes adaptées qui permettent de contrôler la situation. Nous pouvons aller encore plus loin dans cette démarche, puisque l'utilisateur ne voit plus du tout l'attribut, il peut même ne pas exister réellement. C'est ce que nous découvrons dans l'exemple suivant :

complexe.py




```
from math import *

class Complexe:
   def __init__(self, réel=0.0, imaginaire=0.0):
      self.réel = réel                  # attribut 'réel'
      self.imaginaire = imaginaire      # attribut 'imaginaire'

   @property
   def module(self): return sqrt(self.réel**2 + self.imaginaire**2)

   @module.setter
   def module(self, nouveau):
      argument = self.argument
      self.réel = nouveau * cos(argument)
      self.imaginaire = nouveau * sin(argument)

   @property
   def argument(self):
      if self.imaginaire == 0: return 0
      if self.réel == 0:
         if self.imaginaire >= 0: return pi/2
         else: return -pi/2
      return atan(self.imaginaire/self.réel)

   @argument.setter
   def argument(self, nouveau):
      module = self.module
      self.réel = module * cos(nouveau)
      self.imaginaire = module * sin(nouveau)
```

principal.py

```
from complexe import Complexe
from math import pi

z = Complexe(3, 4)
print('module =', z.module)
print('argument =', z.argument)

z.module = 10
print('réel =', z.réel)
print('imaginaire =', z.imaginaire)

z.argument = pi/4
print('réel =', z.réel)
print('imaginaire =', z.imaginaire)
```

Shell Python : [évaluer principal.py]

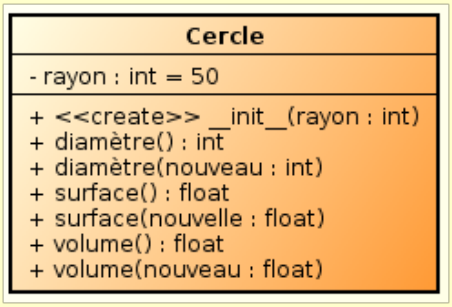
```
module = 5.0
argument = 0.9272952180016122
réel = 6.000000000000001
imaginaire = 7.999999999999999
réel = 7.0710678118654755
imaginaire = 7.071067811865475
```

*L'intervention sur les propriétés **« module »** et **« argument »** ont une conséquence directe sur les attributs effectifs **« réel »** et **« imaginaire »**, alors que les attributs **« module »** et **« argument »** n'existent pas vraiment.*

Au travers de cet exemple, je pense que la démonstration est faite sur l'intérêt capital d'avoir des méthodes « mutateurs ». Elles permettent de bien accompagner le changement d'état de l'objet (évolution des attributs).

Afin de valider ces différents concepts, je vous propose de prendre un dernier exemple qui nous permet de résoudre les différents traitements associés à la notion de cercle : rayon, diamètre, surface d'un disque et volume d'une sphère.

cercle.py




```
from math import pi, sqrt, pow

class Cercle:

    # Définition de l'attribut 'rayon'
    def __init__(self, rayon=50):
        self.rayon = rayon

    # Définition du diamètre du cercle
    @property
    def diamètre(self): return 2 * self.rayon
    @diamètre.setter
    def diamètre(self, nouveau): self.rayon = nouveau//2

    # Définition de la surface d'un disque
    @property
    def surface(self): return pi * self.rayon**2
    @surface.setter
    def surface(self, nouvelle): self.rayon = sqrt(nouvelle/pi)

    # Définition du volume d'une sphère
    @property
    def volume(self): return 4/3 * pi * self.rayon**3
    @volume.setter
    def volume(self, nouveau): self.rayon = pow(3*nouveau/(4*pi), 1/3)
```

principal.py

```
from cercle import Cercle

c1 = Cercle()
c2 = Cercle(70)

trame = '{} : Rayon={:0.0f}m, Diamètre={:0.0f}m, Surface={:0.0f}m², Volume={:0.0f}m³'
print(trame.format(1, c1.rayon, c1.diamètre, c1.surface, c1.volume))
print(trame.format(2, c2.rayon, c2.diamètre, c2.surface, c2.volume))

c1.diamètre = 80
print(trame.format(1, c1.rayon, c1.diamètre, c1.surface, c1.volume))

c1.surface = 3000
print(trame.format(1, c1.rayon, c1.diamètre, c1.surface, c1.volume))

c1.volume = 460000
print(trame.format(1, c1.rayon, c1.diamètre, c1.surface, c1.volume))
```

Shell Python : [évaluer principal.py]

```
1 : Rayon=50m, Diamètre=100m, Surface=7854m², Volume=523599m³
2 : Rayon=70m, Diamètre=140m, Surface=15394m², Volume=1436755m³
1 : Rayon=40m, Diamètre=80m, Surface=5027m², Volume=268083m³
1 : Rayon=31m, Diamètre=62m, Surface=3000m², Volume=123608m³
1 : Rayon=48m, Diamètre=96m, Surface=7204m², Volume=460000m³
```

Attributs de classe

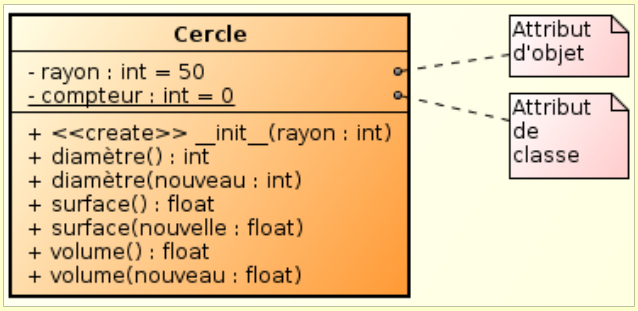
Les attributs sur lesquels nous avons travaillé jusqu'à présent sont appelés des attributs d'objets. Chaque objet possède ainsi ses propres valeurs d'attribut. Le meilleur exemple, est l'identité d'une personne. Chaque personne possède chacune sa propre identité qui caractérise ainsi l'objet associé.

Dans certains cas, il peut être intéressant d'avoir une information commune à tous les objets créés, une information associée du coup à la classe et non plus à un objet en particulier. C'est ce que nous appelons un attribut de classe. Dans ce cadre là, tous les objets créés possèdent exactement la même valeur. Dès qu'une modification apparaît sur l'attribut de classe, tous les objets en sont automatiquement informés.

*À titre d'exemple, je vous propose de rajouter un attribut de classe **« compteur »** sur la classe **« Cercle »** qui nous renseigne en temps réel sur le nombre de cercles créés au moment de la consultation :*

cercle.py

```
from math import pi, sqrt, pow

class Cercle:
    # Création de l'attribut de classe 'compteur'
    compteur = 0

    # Constructeur
    def __init__(self, rayon=50):
        self.rayon = rayon
        Cercle.compteur += 1

    # Définition du diamètre du cercle
    @property
    def diamètre(self): return 2 * self.rayon
    @diamètre.setter
    def diamètre(self, nouveau): self.rayon = nouveau//2

    # Définition de la surface d'un disque
    @property
    def surface(self): return pi * self.rayon**2
    @surface.setter
    def surface(self, nouvelle): self.rayon = sqrt(nouvelle/pi)

    # Définition du volume d'une sphère
    @property
    def volume(self): return 4/3 * pi * self.rayon**3
    @volume.setter
    def volume(self, nouveau): self.rayon = pow(3*nouveau/(4*pi), 1/3)
```



principal.py

```
from cercle import Cercle

motif = '{} : Rayon={}, nombre de cercles = {}'

c1 = Cercle()
c2 = Cercle(70)
print(motif.format('c1', c1.rayon, c1.compteur))
print(motif.format('c2', c2.rayon, c2.compteur))

c3 = Cercle(rayon=150)
print(motif.format('c3', c3.rayon, c1.compteur))

c4 = Cercle()
print(motif.format('c4', c4.rayon, Cercle.compteur))
```

Shell Python : [évaluer principal.py]

```
c1 : Rayon=50, nombre de cercles = 2
c2 : Rayon=70, nombre de cercles = 2
c3 : Rayon=150, nombre de cercles = 3
c4 : Rayon=50, nombre de cercles = 4
```

Un attribut de classe se déclare, bien sûr à l'intérieur de la classe dont il fait parti, au tout début, en dehors de toutes méthodes, contrairement aux attributs d'objet qui sont eux déclarés dans le constructeur.

*Comme il fait parti de la classe, lorsque vous devez réaliser un traitement spécifique sur un attribut de classe, vous devez le préfixer avec le nom de la classe suivi de l'opérateur de séparation classique **« . »**. Nous n'utilisons plus le préfixe **« self »**, puisque ce dernier est associé à l'objet en cours, celui qui réalise un traitement qui lui est propre.*

*Lorsque vous désirez connaître la valeur de l'attribut de classe, vous avez deux solutions, soit vous le préfixez avec le nom de l'objet sur lequel vous travaillez actuellement (ou un autre puisque la valeur est similaire), ou bien vous pouvez choisir le préfixe avec le nom de la classe. Nous avons utilisé les deux options dans le script **« principal.py »**.*

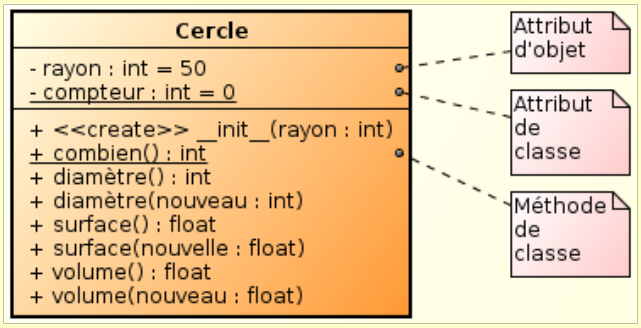
Méthodes de classe

Le script précédent fonctionne très bien. Toutefois, nous sommes dans la même problématique que pour les attributs d'objets, l'attribut de classe **« compteur »** est certes accessible de l'extérieur, mais nous pouvons également le modifier, ce qui est très préjudiciable pour cette application.

Nous sommes donc obligé de passer par un attribut de classe privé, et donc par une méthode spécifique, en lecture seule, qui permet de renseigner l'état du ***« compteur »*** *actuel.*

De même qu'il existe des attributs de classe, il existe aussi des méthodes de classe. Ces méthodes se définissent exactement comme les méthodes classiques, à la différence qu'elles ne prennent pas en premier paramètre l'élément « self » (l'instance de l'objet), mais un autre nommé « cls » (qui représente la classe). Enfin, la déclaration d'une telle méthode est préfixé par le décorateur prédéfini « @classmethod ».

cercle.py




```
from math import pi, sqrt, pow

class Cercle:
    # Création de l'attribut de classe 'compteur'
    __compteur = 0

    # Constructeur
    def __init__(self, rayon=50):
        self.rayon = rayon
        Cercle.__compteur += 1

    # Définition de la méthode de classe
    @classmethod
    def combien(cls): return cls.__compteur

    # Définition du diamètre du cercle
    @property
    def diamètre(self): return 2 * self.rayon
    @diamètre.setter
    def diamètre(self, nouveau): self.rayon = nouveau//2

    # Définition de la surface d'un disque
    @property
    def surface(self): return pi * self.rayon**2
    @surface.setter
    def surface(self, nouvelle): self.rayon = sqrt(nouvelle/pi)

    # Définition du volume d'une sphère
    @property
    def volume(self): return 4/3 * pi * self.rayon**3
    @volume.setter
    def volume(self, nouveau): self.rayon = pow(3*nouveau/(4*pi), 1/3)
```

principal.py

```
from cercle import Cercle

motif = '{} : Rayon={}, nombre de cercles = {}'

c1 = Cercle()
c2 = Cercle(70)
print(motif.format('c1', c1.rayon, c1.combien()))
print(motif.format('c2', c2.rayon, c2.combien()))

c3 = Cercle(rayon=150)
print(motif.format('c3', c3.rayon, c1.combien()))

c4 = Cercle()
print(motif.format('c4', c4.rayon, Cercle.combien()))

Cercle.compteur = 0
print(motif.format('c4', c4.rayon, Cercle.combien()))
```

Shell Python : [évaluer principal.py]

```
c1 : Rayon=50, nombre de cercles = 2
c2 : Rayon=70, nombre de cercles = 2
c3 : Rayon=150, nombre de cercles = 3
c4 : Rayon=50, nombre de cercles = 4
c4 : Rayon=50, nombre de cercles = 4
```

Les méthodes spéciales

Les méthodes spéciales sont des méthodes bien particulières puisqu'elles réalisent des traitements de façon automatique sans que nous y fassions appel explicitement. Nous en avons déjà rencontrée une, le constructeur, nommée ici **« __init__ »,** qui est implicitement utilisée (aucun appel explicite), lors de la création d'un nouvel objet. Toutes les méthodes spéciales dans Python sont reconnaissables par le double souligné **« __ »** de part et d'autre du nom de la méthode.

Je vous propose un tableau non exhaustif de la plupart des méthodes spéciales qui peuvent être exploitées dans vos différentes classes afin d'enrichir leurs comportements. Nous ne les passerons pas toutes en revue, mais nous en découvrirons certaines afin de bien voir comment les définir et les utiliser par la suite.

Catégorie	Nom des méthodes spéciales
Représentation	*__repr__, __str__, __format__, __bytes__*
Conversion en nombre	*__bool__, __int__, __float__, __index__*
Collections	*__len__, « [ ] » __getitem__, « [ ] »__setitem__, __delitem__, « in »__contains__*
Itérateurs	*« for … in »__iter__, __reversed__, __next__*
Création et destruction	*__new__, __init__ (constructeur), __del__(destructeur)*
Gestion des attributs	*__getattr__, __setattr__, __delattr__, __dir__*
Comparaison	*« < » __lt__, « <= » __le__, « == » __eq__, «!= » __ne__, « > »__gt__, « >= » __ge__*
Opérateurs arithmétiques	*« + » __add__, « - » __sub__, « * » __mul__, « / » __truediv__, « // » __floordiv__, « % » __mod__, « + » __radd__, etc., « += » __iadd__, etc., « ** » __pow__*

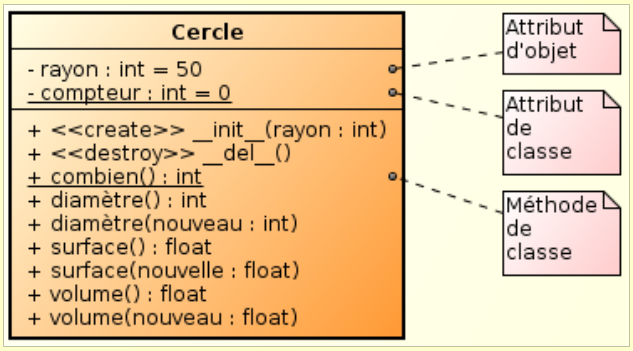
Proposition d'un destructeur sur la classe « Cercle »

Précédemment, dans la classe **« Cercle »**, nous avons rajouté un attribut de classe **« compteur »** qui nous renseigne, en temps réel, sur le nombre d'objets en cours de fonctionnement. À chaque nouvel objet créé, cet attribut est automatiquement incrémenté.

Que se passe-t-il si nous détruisons un objet cercle ? Pour l'instant, aucun fonctionnement adapté n'est prévu pour gérer cette situation. L'idéal serait de prévoir un destructeur qui va permettre de décrémenter automatiquement cet attribut de classe.

cercle.py

```
from math import pi, sqrt, pow

class Cercle:
    # Création de l'attribut de classe 'compteur'
    __compteur = 0

    # Constructeur
    def __init__(self, rayon=50):
        self.rayon = rayon
        Cercle.__compteur += 1

    # Destructeur
    def __del__(self): Cercle.__compteur -= 1

    # Définition de la méthode de classe
    @classmethod
    def combien(cls): return cls.__compteur

    # Définition du diamètre du cercle
    @property
    def diamètre(self): return 2 * self.rayon
    @diamètre.setter
    def diamètre(self, nouveau): self.rayon = nouveau//2

    # Définition de la surface d'un disque
    @property
    def surface(self): return pi * self.rayon**2
    @surface.setter
    def surface(self, nouvelle): self.rayon = sqrt(nouvelle/pi)

    # Définition du volume d'une sphère
    @property
    def volume(self): return 4/3 * pi * self.rayon**3
    @volume.setter
    def volume(self, nouveau): self.rayon = pow(3*nouveau/(4*pi), 1/3)
```



principal.py

```
from cercle import Cercle
motif = '{} : Rayon={}, nombre de cercles = {}'
c1 = Cercle()
c2 = Cercle(70)
print(motif.format('c1', c1.rayon, c1.combien()))
print(motif.format('c2', c2.rayon, c2.combien()))
```

```
c3 = Cercle(rayon=150)
print(motif.format('c3', c3.rayon, c1.combien()))

Cercle.compteur = 0
del(c2)
print(motif.format('c1', c1.rayon, c1.combien()))
```

Shell Python : [évaluer principal.py]

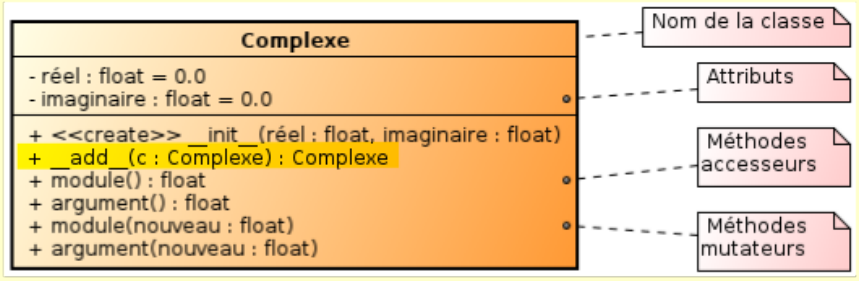
```
c1 : Rayon=50, nombre de cercles = 2
c2 : Rayon=70, nombre de cercles = 2
c3 : Rayon=150, nombre de cercles = 3
c1 : Rayon=50, nombre de cercles = 2
```

*La méthode « **del()** », avec laquelle nous avons déjà travaillé, permet de supprimer n'importe quel type d'objet. Par contre, l'objet en question doit justement posséder un destructeur spécifique qui permet de savoir quoi faire, ici **« __del__ »**.*

Redéfinition des opérateurs – addition de deux nombres complexes

Python, à l'image du C++, est vraiment très riche, grâce notamment aux méthodes spéciales, puisqu'il permet de proposer d'autres comportements pour les opérateurs, comme par exemple, permettre l'addition entre deux nombres complexes.

complexe.py




```
from math import *

class Complexe:
   def __init__(self, réel=0.0, imaginaire=0.0):
      self.réel = réel                  # attribut 'réel'
      self.imaginaire = imaginaire      # attribut 'imaginaire'

   # Addition de deux nombres complexes
   def __add__(self, c):
      z = Complexe()
      z.réel = self.réel + c.réel
      z.imaginaire = self.imaginaire + c.imaginaire
      return z

   @property
   def module(self): return sqrt(self.réel**2 + self.imaginaire**2)

   @module.setter
   def module(self, nouveau):
      argument = self.argument
      self.réel = nouveau * cos(argument)
      self.imaginaire = nouveau * sin(argument)

   @property
   def argument(self):
      if self.imaginaire == 0: return 0
      if self.réel == 0:
         if self.imaginaire >= 0: return pi/2
         else: return -pi/2
      return atan(self.imaginaire/self.réel)

   @argument.setter
   def argument(self, nouveau):
      module = self.module
      self.réel = module * cos(nouveau)
      self.imaginaire = module * sin(nouveau)
```

principal.py

```
from complexe import Complexe

# Création de plusieurs objets a, b, c de la classe 'Complexe'
a = Complexe(5.3, 8.9)
b = Complexe(2.5)
```

```
c = Complexe()
c.imaginaire = -1
d = a + b + c

# Affichage du contenu des différents objets
motif = '{} : (réel={}, imaginaire={})'
print(motif.format('a', a.réel, a.imaginaire))
print(motif.format('b', b.réel, b.imaginaire))
print(motif.format('c', c.réel, c.imaginaire))
print(motif.format('d', d.réel, d.imaginaire))
```

Shell Python : [évaluer principal.py]

```
a : (réel=5.3, imaginaire=8.9)
b : (réel=2.5, imaginaire=0.0)
c : (réel=0.0, imaginaire=-1)
d : (réel=7.8, imaginaire=7.9)
```

*Pour redéfinir l'opérateur d'addition pour les nombres complexes, vous devez simplement rajouter dans votre classe **« Complexe »**, la méthode spéciale **« __add__ »**. L'addition, je le rappelle, prend uniquement deux opérandes, ici des nombres complexes, dont l'un est le premier objet **« self »** qui lance la méthode.*

Lorsque vous écrivez « z = a + b », cela équivaut en réalité à l'écriture suivante : « z = a.__add__(b) ». Cette deuxième écriture est bien sûr moins jolie, mais elle est totalement équivalente. Lorsque dans le script vous proposer la première écriture, l'interpréteur de Python va bien contrôler s'il existe bien une méthode interne « __add__() » associée à cet objet « a ».

Cette méthode renvoie bien entendu un nombre complexe. Pour réaliser le calcul, vous remarquez que nous avons besoin de déclarer également un nombre complexe qui va servir de résultat intermédiaire. C'est ce nombre complexe qui sera renvoyé par la méthode.

Addition entre des nombres complexes et des nombres réels

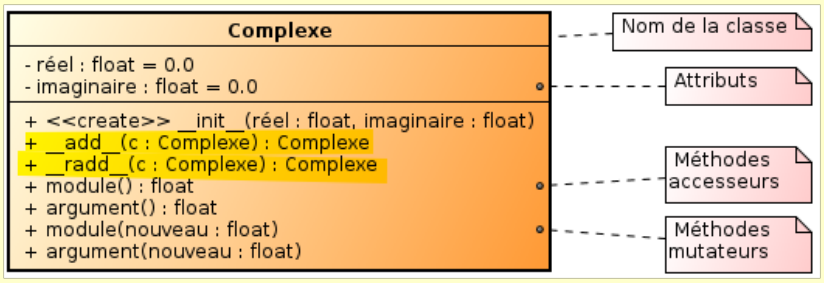
Que se passe-t-il si nous faisons la somme d'un nombre complexe avec un nombre réel ? Cela est tout à fait possible en mathématique puisque les nombres réels font parti des nombres complexes. Par contre ici, si vous tentez l'expérience, une erreur sera produite puisque dans notre méthode nous attendons des nombres complexes en arguments pour discriminer la partie réelle de la partie imaginaire.

Pour résoudre ce problème, il suffit de vérifier le type de l'argument supplémentaire et de fabriquer un nombre complexe en conséquence si ce dernier est un réel ou un entier.

*Par ailleurs, normalement, l'addition est commutative, sauf que si vous placez le nombre réel à gauche de l'opérateur **« + »**, l'interpréteur vérifiera alors si le nombre réel possède bien la méthode **« __add__ »** avec pour paramètre un nombre complexe. Sauf que par défaut, ce paramètre n'existe pas puisque la classe **« Complexe »** n'est pas connue par le type **« float »** puisque nous l'avons construite de toute pièce.*

Pour résoudre ce dernier cas de figure, il existe une méthode spéciale qui permet éventuellement de lire l'opération dans l'autre sens. Pour l'addition, il s'agit de la méthode « __radd__ », qui signifie « reverse add ».

complexe.py




```
from math import *

class Complexe:
   def __init__(self, réel=0.0, imaginaire=0.0):
      self.réel = réel                  # attribut 'réel'
      self.imaginaire = imaginaire      # attribut 'imaginaire'

   # Addition de deux nombres complexes
   def __add__(self, c):
      if type(c) is float or type(c) is int : c = Complexe(réel=c)
      return Complexe(self.réel+c.réel, self.imaginaire+c.imaginaire)

   # Addition de deux nombres complexes en sens inverse
   def __radd__(self, valeur): return self+valeur # appel de la méthode précédente

 . . .
```

principal.py

```
from complexe import Complexe

# Création de plusieurs objets a, b, c de la classe 'Complexe'
```

```
a = Complexe(5.3, 8.9)
b = Complexe(2.5)
c = Complexe()
c.imaginaire = -1
d = a + b + c

# Affichage du contenu des différents objets
motif = '{} : (réel={}, imaginaire={})'
print(motif.format('a', a.réel, a.imaginaire))
print(motif.format('b', b.réel, b.imaginaire))
print(motif.format('c', c.réel, c.imaginaire))
print(motif.format('d', d.réel, d.imaginaire))

c = d + 5.2
a = 3 + c
print(motif.format('c', c.réel, c.imaginaire))
print(motif.format('a', a.réel, a.imaginaire))
```

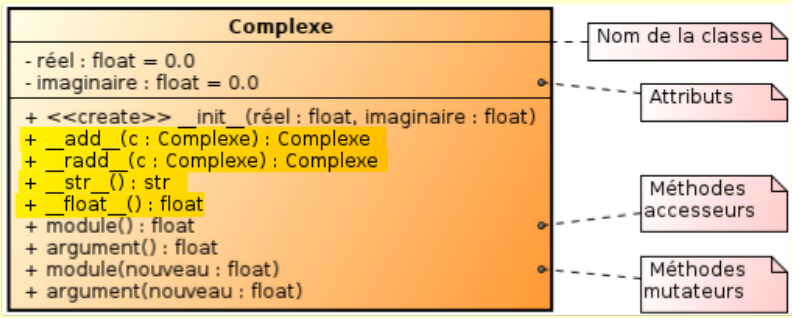
Shell Python : [évaluer principal.py]

```
a : (réel=5.3, imaginaire=8.9)
b : (réel=2.5, imaginaire=0.0)
c : (réel=0.0, imaginaire=-1)
d : (réel=7.8, imaginaire=7.9)
```

Transformation : Complexe → chaîne de caractère - Complexe → réel

Il est également possible de prévoir des transformations automatiques vers d'autres types de variable, par exemple transformer un nombre complexe vers une chaîne de caractères afin que sa visualisation soit plus intuitive et immédiate. Nous pouvons prévoir également une transformation d'un complexe vers un réel sachant que dans ce cas, c'est la partie réelle qui est renvoyée automatiquement.

complexe.py




```
from math import *

class Complexe:
   def __init__(self, réel=0.0, imaginaire=0.0):
      self.réel = réel                # attribut 'réel'
      self.imaginaire = imaginaire    # attribut 'imaginaire'

   # Addition de deux nombres complexes
   def __add__(self, c):
      if type(c) is float or type(c) is int : c = Complexe(réel=c)
      return Complexe(self.réel+c.réel, self.imaginaire+c.imaginaire)

   # Addition de deux nombres complexes en sens inverse
   def __radd__(self, valeur): return self+valeur # appel de la méthode précédente

   # Transformer en chaîne de caractères lisible
   def __str__(self):
      return "(réel={}, imaginaire={})".format(self.réel, self.imaginaire)

   # Récupérer la partie réelle d'un nombre complexe
   def __float__ (self): return self.réel

...
```

principal.py

```
from complexe import Complexe

# Création de plusieurs objets a, b, c de la classe 'Complexe'
a = Complexe(5.3, 8.9)
b = Complexe(2.5)
c = Complexe()
c.imaginaire = -1
```

```
d = a + b + c

# Affichage du contenu des différents objets
print('a', a)
print('b', b)
print('c', c)
print('d', d)

c = d + 5.2
a = 3 + c
réel = float(a)

print('c', c)
print('a', a)
print('réel =', réel)
```

Shell Python : [évaluer principal.py]

```
a (réel=5.3, imaginaire=8.9)
b (réel=2.5, imaginaire=0.0)
c (réel=0.0, imaginaire=-1)
d (réel=7.8, imaginaire=7.9)
c (réel=13.0, imaginaire=7.9)
a (réel=16.0, imaginaire=7.9)
réel = 16.0
```

Classe de gestion de notes : d'autres méthodes spéciales

Afin de conclure sur le chapitre des méthodes spéciales, je vous propose de créer une nouvelle classe **« Notes »** qui permet de gérer un certain nombre d'actions associées, comme le calcul de la valeur moyenne, retrouver la note maxi, la note mini, le nombre de notes, la présence d'une note dans la liste, etc.

notes.py

```
class Notes:
    def __init__(self, notes=[]):
        self.__notes = notes

    def ajout(self, note):
        if type(note) is list: self.__notes.extend(note)
        else: self.__notes.append(note)

    def moyenne(self):
        nombre = len(self.__notes)
        if nombre==0: return 0
        somme = 0
        for note in self.__notes: somme += note
        return somme/nombre

    def maxi(self):
        maximum=0
        for note in self.__notes:
            if note>maximum: maximum=note
        return maximum

    def mini(self):
        if len(self.__notes)==0: return 0
        minimum=20
        for note in self.__notes:
            if note<minimum: minimum=note
        return minimum

    def __len__(self): return len(self.__notes)
    def __getitem__(self, index): return self.__notes[index]
    def __contains__(self, note): return note in self.__notes

    def __iter__(self):
        for note in self.__notes:
            yield note

    def __str__(self):
        motif = "{}, moyenne={:0.1f}, maxi={}, mini={}"
        return motif.format(self.__notes, self.moyenne(), self.maxi(), self.mini())
```

principal.py

```
from notes import Notes
```

```
élève1 = Notes([18.5, 8.5, 12, 15, 7.5])
élève2 = Notes()
élève2.ajout([12.5, 10, 9, 15])
élève1.ajout(9);

print(élève1)
print(élève2)
print('Nombre de notes de élève1 :', len(élève1))
print('Nombre de notes de élève2 :', len(élève2))
print('Première note de élève1 :', élève1[0])
print('Dernière note de élève2 :', élève2[-1])
print('La note "8.5" existe-t-elle dans élève1 :', 8.5 in élève1)
print('La note "8.5" existe-t-elle dans élève2 :', 8.5 in élève2)
print("Toutes les notes d'élève1 = ", end=' ')
for note in élève1: print(note, end=' ')
```

Shell Python : [évaluer principal.py]

```
[18.5, 8.5, 12, 15, 7.5, 9], moyenne=11.8, maxi=18.5, mini=7.5
[12.5, 10, 9, 15], moyenne=11.6, maxi=15, mini=9
Nombre de notes de élève1 : 6
Nombre de notes de élève2 : 4
Première note de élève1 : 18.5
Dernière note de élève2 : 15
La note "8.5" existe-t-elle dans élève1 : True
La note "8.5" existe-t-elle dans élève2 : False
Toutes les notes d'élève1 =  18.5 8.5 12 15 7.5 9
```

*Dans la méthode **« __iter__ »**, vous remarquez la présence du mot réservé **« yield »** en lieu et place de **« return »**. Ce mot clé est similaire au **« return »** utilisé dans les méthodes sauf qu'il ne signifie pas la fin de l'exécution de la méthode mais une mise en pause et à la prochaine itération la méthode recherchera le prochain **« yield »**.*

Héritage et polymorphisme

Lorsque nous consultons le code précédent, vous remarquez que le nom des objets choisis sont **« élève1 »** et **<élève2 »**. Certes, ces objets sont bien associés à la notion de notes, mais des élèves sont aussi des personnes avec chacun une identité propre. Ce qui serait plus judicieux, serait d'associer cette notion de personne à cette notion de notes.

*L'idéal ici n'est pas de tout reconstruire, mais de bénéficier ainsi des travaux réaliser lors de la conception de la classe **« Personne »**. Un élève est un cas particulier d'une personne, il a un nom et un prénom, mais il possède en plus des notes.*

La meilleure technique pour cela consiste à réaliser un héritage, c'est-à-dire de créer une nouvelle classe « Élève » qui hérite de la classe « Personne », et de rajouter dans cette classe tout ce qui concerne un élève, c'est-à-dire des méthodes supplémentaires pour la gestion des notes, comme nous venons de le faire dans le chapitre précédent.

*Grâce à l'héritage, la classe **« Élève »** récupère automatiquement toutes les méthodes de la classe **« Personne »** par héritage. Toutes ces méthodes héritées font maintenant parties de la classe **« Élève »** comme si nous les avions explicitement écrites. Il vous suffit ensuite de décrire toutes les méthodes supplémentaires qui caractérisent un élève.*

La syntaxe pour réaliser un héritage est très très simple, il suffit de spécifier le nom de votre classe, de placer ensuite entre parenthèses le nom de votre classe (parente) dont vous voulez hériter, et c'est fini. Il est possible d'hériter de plusieurs classes. Vous les séparez par l'opérateur « , ». Dans ce cas, il s'agit d'un héritage multiple.

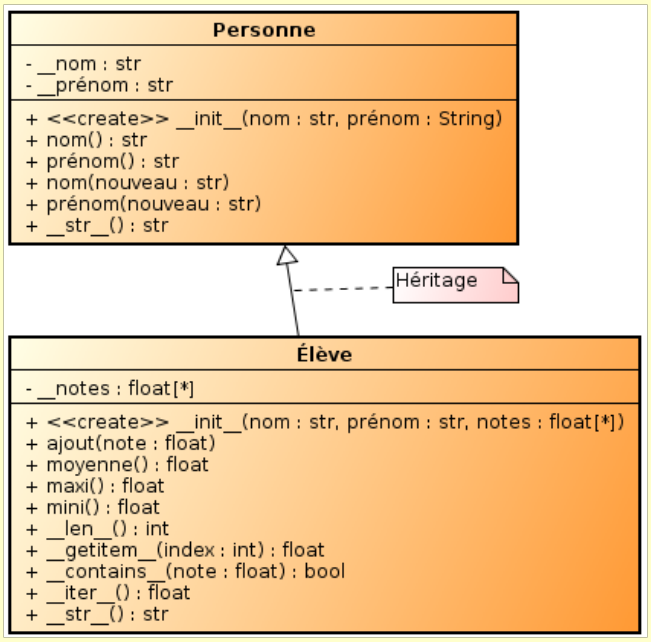
personne.py

```
class Personne:
   def __init__(self, nom, prénom):
      self.__nom = nom.upper()
      self.__prénom = prénom.title()

   @property
   def nom(self): return self.__nom
   @nom.setter
   def nom(self, nouveau): self.__nom = nouveau.upper()

   @property
   def prénom(self): return self.__prénom
   @prénom.setter
   def prénom(self, nouveau): self.__prénom = nouveau.title()

   def __str__(self): return self.__prénom + ' ' + self.__nom
```

eleve.py

```
from personne import Personne

# la classe 'Élève' hérite de la classe 'Personne'
class Élève(Personne):
    def __init__(self, nom, prénom, notes=[]):
```

```
        super().__init__(nom, prénom)  # appel au constructeur de la classe parente (super-classe)
        self.__notes = notes           # ou bien "Personne.__init__(self, nom, prénom)"

    def ajout(self, note):
        if type(note) is list: self.__notes.extend(note)
        else: self.__notes.append(note)

    @property
    def moyenne(self):
        nombre = len(self.__notes)
        if nombre==0: return 0
        somme = 0
        for note in self.__notes: somme += note
        return somme/nombre

    @property
    def maxi(self):
        maximum=0
        for note in self.__notes:
            if note>maximum: maximum=note
        return maximum

    @property
    def mini(self):
        if len(self.__notes)==0: return 0
        minimum=20
        for note in self.__notes:
            if note<minimum: minimum=note
        return minimum

    def __len__(self): return len(self.__notes)
    def __getitem__(self, index): return self.__notes[index]
    def __contains__(self, note): return note in self.__notes
    def __iter__(self):
        for note in self.__notes:
            yield note

    def __str__(self):
        motif = "{} : {}, moyenne={:0.1f}, maxi={}, mini={}"
        return motif.format(super().__str__(), self.__notes, self.moyenne, self.maxi, self.mini)
```



principal.py

```
from eleve import Élève

élève1 = Élève('durand', 'michel', [18.5, 8.5, 12, 15, 7.5])
élève2 = Élève('dupond', 'alice')
élève2.ajout([12.5, 10, 9, 15])
élève1.ajout(9);

print(élève1)
print(élève2)
print('Nombre de notes de', élève1.prénom, ':', len(élève1))
print('Nombre de notes de', élève2.prénom, ':', len(élève2))
print('Première note de', élève1.prénom, ':', élève1[0])
print('Dernière note de', élève2.prénom, ':', élève2[-1])
print(élève1.prénom, 'possède t-il la note "8.5":', 8.5 in élève1)
print(élève2.prénom, 'possède t-elle la note "8.5":', 8.5 in élève2)
print("Toutes les notes de", élève1.prénom, '=>', end=' ')
for note in élève1: print(note, end=' ')
```

Shell Python : [évaluer principal.py]

```
Michel DURAND : [18.5, 8.5, 12, 15, 7.5, 9], moyenne=11.8, maxi=18.5, mini=7.5
Alice DUPOND : [12.5, 10, 9, 15], moyenne=11.6, maxi=15, mini=9
Nombre de notes de Michel : 6
Nombre de notes de Alice : 4
Première note de Michel : 18.5
Dernière note de Alice : 15
Michel possède t-il la note "8.5": True
Alice possède t-elle la note "8.5": False
Toutes les notes de Michel => 18.5 8.5 12 15 7.5 9
```

Lorsque vous créez votre constructeur sur la classe enfant, vous n'avez pas à initialiser tous les attributs dont vous héritez, c'est déjà fait, il suffit alors d'appeler explicitement le constructeur de la classe parente en utilisant la méthode ***« super() »*** *suivie du nom de la méthode qui nous intéresse, c'est-à-dire ici* ***« __init()__ ».*** *Nous avons utiliser la même technique lors de l'élaboration de la méthode* ***« __str__ ».***