Programmation orientée objet (POO) : objets, classes, méthodes, instances, héritage, polymorphisme

Auteur: Albert Guedj, Gymnase du Bugnon, 3MOCINFO

Date: 26.05.2018

Sources et références:

- Le tutoriel Python: chapitre 9 - Classes (https://docs.python.org/fr/3/tutorial/classes.html) par la Python Software Foundation
- Think Python 2nd edition (https://greenteapress.com/wp/think-python-2e/) par Allen B. Downey
- Pensez en Python (https://allen-downey.developpez.com/livres/python/pensez-python/) traduction en français
- Programmation orientée objet (POO) (https://fr.wikipedia.org/wiki/Programmation_orientée_objet) sur Wikipédia

La **programmation orientée objet** (**POO**), ou programmation par objet, est un paradigme de programmation informatique qui utilise la définition et l'interaction de briques logicielles appelées **objets**. Il s'agit donc de représenter ces **objets** et leurs relations ; l'interaction entre les **objets** via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la **POO**. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

Un **objet** représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs.

Une **classe** est un moyen de réunir des données et des fonctionnalités. Créer une nouvelle **classe** crée un nouveau type d'**objet** et ainsi de nouvelles **instances** de ce type peuvent être construites. Chaque **instance** peut avoir ses propres **attributs**, ce qui définit son état. Une **instance** peut aussi avoir des **méthodes** (définies par la classe de l'instance) pour modifier son état.

Classes, objets et instances

Une **classe** est un type défini par le programmeur et voici sa déifinition ci-dessous:

In [199]:

```
class Point:
    """Représente un point dans l'espace 2-D."""
```

L'en-tête indique que la nouvelle classe s'appelle Point. Le corps est une docstring qui explique à quoi sert la classe.

Instanciation

L'**objet classe** est une espèce d'usine à créer des objets. Pour créer un Point, vous appelez Point comme si

c'était une fonction.
La création d'un nouvel objet s'appelle **instanciation**, et l'objet est une **instance** de la classe.

In [200]:

```
pt = Point()
```

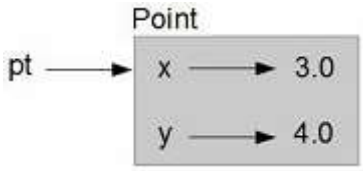
Attributs

Pour attribuer des valeurs à une instance il faut utiliser la notation ci-dessous:

In [201]:

```
pt.x = 3.0
pt.y = 4.0
```

Le schéma suivant montre le résultat de ces affectations. Un diagramme d'état qui montre un objet et ses attributs s'appelle un **diagramme d'objets**



Pour lire la valeur d'un attribut il suffit d'utiliser la même syntaxe:

In [202]:

```
pt.x
```

Out[202]:

```
3.0
```

Instances

Une classe peut aussi utiliser une instance d'une autre classe comme attribut. Par exemple la classe rectangle ci-dessous:

In [203]:

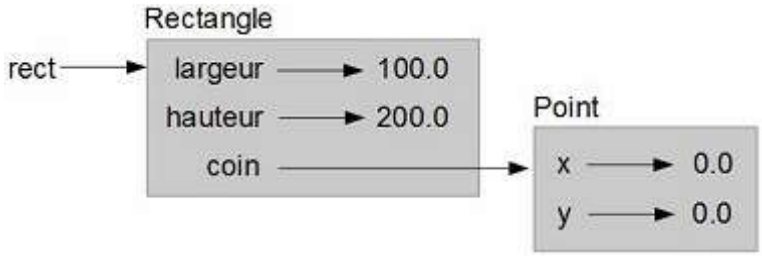
```
class Rectangle:
    """Représente un rectangle.
    attributs: largeur, hauteur, coin.
    """
```

Le docstring répertorie les attributs : largeur et hauteur sont des nombres ; coin est un objet Point qui spécifie le coin inférieur gauche.

In [204]:

```
rect = Rectangle()
rect.largeur = 100.0
rect.hauteur = 200.0
rect.coin = Point()
rect.coin.x = 0.0
rect.coin.y = 0.0
```

L'expression `rect.coin.x` signifie, « Va à l'objet auquel se réfère rect et sélectionne l'attribut nommé coin ; puis va à cet objet et sélectionne l'attribut nommé x. »



Un objet qui est un attribut d'un autre objet est **inclus**

Il est possibble de créer des fonctions renvoyant des instances

In [205]:

```
def trouver_centre(rectangle):
    """Trouve le centre d'un rectangle"""
    p = Point()
    p.x = rectangle.coin.x + rectangle.largeur/2
    p.y = rectangle.coin.y + rectangle.hauteur/2
    return p

centre = trouver_centre(rect)
centre.x, centre.y
```

Out[205]:

```
(50.0, 100.0)
```

Vous pouvez modifier l'état d'un objet en faisant une affectation à l'un de ses attributs. Par exemple, pour modifier la taille d'un rectangle sans modifier sa position, vous pouvez modifier les valeurs de largeur et de hauteur

In [206]:

```
def agrandir_rectangle(rectangle, dLargeur, dHauteur):
    """Aggrandis un rectangle d'une largeur et hauteur donnees"""
    rectangle.largeur += dLargeur
    rectangle.hauteur += dHauteur

agrandir_rectangle(rect, 50, 100)
rect.largeur, rect.hauteur
```

Out[206]:

```
(150.0, 300.0)
```

Copie

Le module copy contient une fonction appelée `copy()` qui peut dupliquer un objet quelconque

In [207]:

```
p1 = Point()
p1.x = 3.0
p1.y = 4.0
import copy
p2 = copy.copy(p1)
```

p1 et p2 contiennent les mêmes données, mais ils ne sont pas le même Point

In [208]:

```
p1 is p2
```

Out[208]:

```
False
```

In [209]:

```
p1 == p2
```

Out[209]:

```
False
```

L'opérateur is indique que p1 et p2 ne sont pas le même objet, comme nous nous y attendions. Mais vous vous attendiez peut-être que l'égalité == soit vraie, parce que ces points contiennent les mêmes données. Dans ce cas, vous serez déçu d'apprendre que pour les instances, le comportement par défaut de l'opérateur `==` est le même que pour l'opérateur `is` ; il vérifie l'identité des objets, pas leur équivalence. Cela arrive parce que, pour les types définis par le programmeur, Python ne sait pas ce qui devrait être considéré comme équivalent. Du moins, pas encore, voir chapitre Méthodes.

Si vous utilisez `copy.copy()` pour dupliquer un Rectangle, vous verrez qu'il copie l'objet Rectangle, mais pas le Point inclus.

In [210]:

```
rect2 = copy.copy(rect)
rect2 is rect
```

Out[210]:

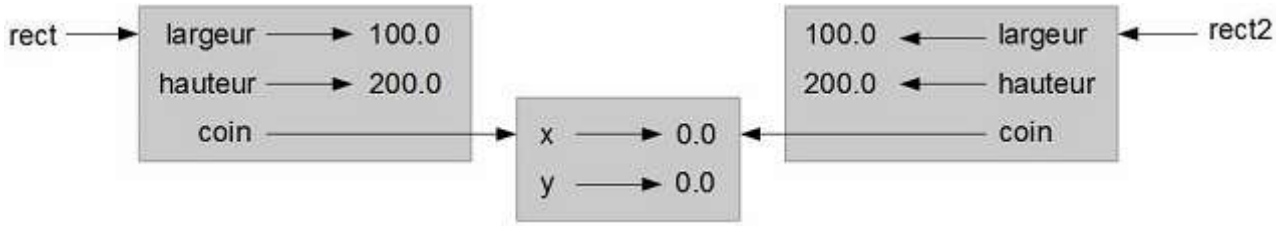
```
False
```

In [211]:

```
rect2.coin is rect.coin
```

Out[211]:

```
True
```



Cette opération s'appelle une **copie superficielle**, car elle copie l'objet et toutes les références qu'il contient, mais pas les objets inclus.

Pour la majorité des applications, ce n'est pas ce que vous voulez. Dans cet exemple, l'invocation de `agrandir_rectangle()` sur l'un des Rectangles n'affecterait pas l'autre, mais l'invocation de `deplacer_rectangle()` sur l'un d'eux affecterait tous les deux ! Ce comportement est source de confusion et d'erreurs.

Heureusement, le module copy fournit une méthode nommée `copy.deepcopy()` qui copie non seulement l'objet, mais aussi les objets auxquels il se réfère, et les objets auxquels ces derniers se réfèrent, et ainsi de suite. Vous ne serez pas surpris d'apprendre que cette opération s'appelle une **copie en profondeur**.

In [212]:

```
rect3 = copy.deepcopy(rect)
rect3 is rect
```

Out[212]:

```
False
```

In [213]:

```
rect3.coin is rect.coin
```

Out[213]:

```
False
```

Classes et fonctions

Afin d'aborder un problème complexe nous pouvons commencer par un simple prototype et en traitant les difficultés de façon incrémentielle, cela s'appelle le **prototypage et correction**.

Par exemple une classe Temps représentant un moment de la journée:

In [214]:

```
class Temps:
    """Représente le moment de la journée.
    attributs : heure, minute, seconde
    """
```

In [215]:

```
def afficher_temps(t):
    """Affiche le temps dans un format traditionel"""
    print(("{}:{}:{}").format(t.heure, t.minute, t.seconde))
```

Si nous voulons créer une fonction qui ne modifie aucun des objets qui lui sont passés comme arguments et qu'elle n'a aucun effet, comme l'affichage d'une valeur ou l'obtention des données saisies par l'utilisateur, autre que de renvoyer une valeur, c'est une **fonction pure**

En général, je vous conseille d'écrire des fonctions pures chaque fois que cela est raisonnable et de recourir à des modificateurs uniquement si cela présente un avantage convaincant. Cette approche pourrait s'appeler un **style fonctionnel de programmation**.

Par exemple, une fonction `ajouter_temps()` qui additionne deux objets temps:

In [216]:

```
def ajouter_temps(t1, t2):
    """Additionne deux temps"""
    somme = Temps()
    somme.heure = t1.heure + t2.heure
    somme.minute = t1.minute + t2.minute
    somme.seconde = t1.seconde + t2.seconde
    return somme
```

In [217]:

```
debut = Temps()
debut.heure = 9
debut.minute = 45
debut.seconde = 0

duree = Temps()
duree.heure = 1
duree.minute = 35
duree.seconde = 0
```

In [218]:

```
fini = ajouter_temps(debut, duree)
afficher_temps(fini)
```

```
10:80:0
```

Le résultat, 10:80:00 n'est peut-être pas ce que vous espériez. Le problème est que cette fonction ne traite pas les cas où le nombre de secondes ou de minutes additionnées dépasse soixante. Lorsque cela se produit, nous devons « retenir » ou « reporter » les secondes supplémentaires dans la colonne des minutes ou les minutes supplémentaires dans la colonne des heures.

Voici une version améliorée :

In [219]:

```
def ajouter_temps2(t1, t2):
    """Additionne deux temps de manière correcte"""
    somme = Temps()
    somme.heure = t1.heure + t2.heure
    somme.minute = t1.minute + t2.minute
    somme.seconde = t1.seconde + t2.seconde
    if somme.seconde >= 60:
        somme.seconde -= 60
        somme.minute += 1
    if somme.minute >= 60:
        somme.minute -= 60
        somme.heure += 1
    return somme
```

In [220]:

```
fini2 = ajouter_temps2(debut, duree)
afficher_temps(fini2)
```

```
11:20:0
```

Parfois, il est utile qu'une fonction puisse modifier les objets qu'elle reçoit comme paramètres. Dans ce cas, les changements sont visibles par la procédure appelante. Ce genre de fonctions s'appellent **modificateurs**.

Par exemple, `incremente()` , qui ajoute un nombre donné de secondes à un objet Temps, peut être écrite naturellement comme un modificateur.

In [221]:

```
def incremente(temps, secondes):
    """Additionne un temps et un nombre de secondes"""
    temps.seconde += secondes
    if temps.seconde >= 60:
        n = temps.seconde//60
        temps.seconde -= n*60
        temps.minute += n
    if temps.minute >= 60:
        m = temps.minute//60
        temps.minute -= m*60
        temps.heure += m
```

In [222]:

```
incremente(debut,200)
afficher_temps(debut)
```

```
9:48:20
```

Une autre possibilité est le **développement par conception**, dans lequel une compréhension de haut niveau du problème peut rendre la programmation beaucoup plus facile. Dans ce cas, il s'agit de comprendre qu'un objet Temps est en fait un nombre à trois chiffres exprimés en base 60

Voici une fonction qui convertit des Temps en entiers :

In [223]:

```
def temps_vers_int(temps):
    """Convertit un temps en nombre de secondes"""
    minutes = temps.heure * 60 + temps.minute
    secondes = minutes * 60 + temps.seconde
    return secondes
```

In [224]:

```
s = temps_vers_int(duree)
s
```

Out[224]:

```
5700
```

Et voici une fonction qui convertit un nombre entier vers un Temps

In [225]:

```
def int_vers_temps(secondes):
    """Convertit un nombre de secondes en temps"""
    temps = Temps()
    minutes, temps.seconde = divmod(secondes, 60)
    temps.heure, temps.minute = divmod(minutes, 60)
    return temps
```

In [226]:

```
duree2 = int_vers_temps(s)
afficher_temps(duree2)
```

```
1:35:0
```

In [227]:

```
def ajouter_temps3(t1, t2):
    """Additionne deux temps de manière optimisee"""
    secondes = temps_vers_int(t1) + temps_vers_int(t2)
    return int_vers_temps(secondes)
```

In [228]:

```
fini3 = ajouter_temps3(debut,duree)
afficher_temps(fini3)
```

```
11:23:20
```

Un objet Temps est bien formé si les valeurs de minute et seconde sont entre 0 et 60 (0 compris, mais 60 non compris) et si heure est positive. heure et minute doivent être des valeurs entières, mais nous pourrions permettre à seconde d'avoir une partie fractionnaire.

De telles exigences s'appellent des **invariants**, parce qu'elles doivent toujours être satisfaites. Autrement dit, si elles ne sont pas vraies, quelque chose a mal tourné.

Écrire du code pour vérifier les invariants peut aider à détecter les erreurs et à trouver leurs causes. Par exemple, vous pourriez avoir une fonction comme `valide_temps()` qui prend un objet Temps et renvoie False si elle enfreint un invariant :

In [229]:

```
def valide_temps(temps):
    """Verifie s'il s'agit d'un temps valide"""
    if temps.heure < 0 or temps.minute < 0 or temps.seconde < 0:
        return False
    if temps.minute >= 60 or temps.seconde >= 60:
        return False
    return True
```

Pour vérifier si les arguments sont valides une **instruction assert** peut être utile, elle vérifie un invariant donné et déclenche une exception si elle échoue :

In [230]:

```
def ajouter_temps_final(t1, t2):
    """Version finale d'ajouter_temps, verifiant la validite des temps"""
    assert valide_temps(t1) and valide_temps(t2)
    secondes = temps_vers_int(t1) + temps_vers_int(t2)
    return int_vers_temps(secondes)
```

In [231]:

```
debut_faux = Temps()
debut_faux.heure = 9
debut_faux.minute = 75
debut_faux.seconde = 0
```

In [232]:

```
#fini_final = ajouter_temps_final(debut_faux,duree)
```

Méthodes

Python est un **langage de programmation orienté objet**, ce qui signifie qu'il offre des fonctionnalités qui prennent en charge la programmation orientée objet, laquelle présente ces caractéristiques déterminantes :

- les programmes incluent des définitions de classes et de méthodes ;
- la plus grande partie du calcul est exprimé en tant qu'opérations sur des objets ;
- les objets représentent souvent des choses dans le monde réel, et les méthodes correspondent souvent à la manière dont les choses du monde réel interagissent.

Une **méthode** est une fonction associée à une classe particulière. Le **sujet** est l'objet sur lequel une méthode est invoquée. Un **argument positionnel** est un argument qui n'inclut pas un nom de paramètre, donc il n'est pas un argument mot-clé.

La méthode `__init__()`

La méthode `__init__()` *(raccourci de "initialisation")* est une méthode spéciale qui est appelée automatiquement lorsqu'un objet est instancié. Son nom complet est *(deux caractères de soulignement, suivis par init, puis deux autres caractères de soulignement)*. Une méthode init pour la classe Point pourrait ressembler à ceci :

In [233]:

```
class Point:
    """Représente un point dans l'espace 2-D."""

    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

Les paramètres sont facultatifs, donc si vous appelez Point sans arguments, vous obtenez les valeurs par défaut.
De plus la méthode init est bien plus efficace que la méthode vue précedemment.

In [234]:

```
p = Point()
p.x
```

Out[234]:

```
0
```

La méthode `__str__()`

La méthode `__str__()` est une méthode spéciale, qui est censée renvoyer une représentation sous forme de chaîne de caractères d'un objet.
Par exemple la méthode `__str__()` ci-dessous renvoie la notation usuelle des points en mathématiques.

In [235]:

```
class Point:
    """Représente un point dans l'espace 2D."""

    def __init__(self, x = 0, y = 0):
        """Initialise un objet Point
        Attributs: x et y
        """
        self.x = x
        self.y = y

    def __str__(self):
        """Retourne les attribut du point dans un format standard"""
        return ("({};{})").format(self.x, self.y)
```

In [236]:

```
p1 = Point(1, 2)
print(p1)
```

```
(1;2)
```

Surcharge d'opérateur

En définissant d'autres méthodes spéciales, vous pouvez spécifier le comportement des opérateurs sur les types définis par le programmeur. Par exemple, si vous définissez une méthode nommée `__add__` pour la classe Temps, vous pouvez utiliser l'opérateur `+` sur les objets de type Temps.

La modification du comportement d'un opérateur afin qu'il fonctionne avec des types définis par le programmeur s'appelle la **surcharge d'opérateur**.

In [237]:

```
class Temps:
    """Représente le moment de la journée.
    attributs : heure, minute, seconde
    """
    def __init__(self, heure = 0, minute = 0, seconde = 0):
        """Initialise un objet Temps

        heure: int
        minute: int
        seconde: int or float
        """
        self.heure = heure
        self.minute = minute
        self.seconde = seconde

    def __str__(self):
        """Retourne un string Temps."""
        return "{}:{}:{}".format(self.heure, self.minute, self.seconde)

    def temps_vers_int(self):
        """Compte les secondes depuis minuit"""
        minutes = self.heure * 60 + self.minute
        secondes = minutes * 60 + self.seconde
        return secondes

    def __add__(self, other):
        """Addition de deux temps"""
        secondes = self.temps_vers_int() + other.temps_vers_int()
        return int_vers_temps(secondes)
```

In [238]:

```
debut = Temps(9, 45)
duree = Temps(1, 35)
print(debut + duree)
```

```
11:20:0
```

Dans la section précédente, nous avons additionné deux objets Temps, mais vous pouvez également additionner un nombre entier à un objet Temps. Ce qui suit est une version de `__add__` qui vérifie le type de other et invoque soit `ajouter_temps()` , soit `incremente()` :

In [239]:

```
class Temps:
    """Représente le moment de la journée.
    attributs : heure, minute, seconde
    """
    def __init__(self, heure = 0, minute = 0, seconde = 0):
        """Initialise un objet Temps

        heure: int
        minute: int
        seconde: int or float
        """
        self.heure = heure
        self.minute = minute
        self.seconde = seconde

    def __str__(self):
        """Retourne un string Temps."""
        return "{}:{}:{}".format(self.heure, self.minute, self.seconde)

    def temps_vers_int(self):
        """Compte les secondes depuis minuit"""
        minutes = self.heure * 60 + self.minute
        secondes = minutes * 60 + self.seconde
        return secondes

    def __add__(self, other):
        """Addition d'un temps et un temps ou nombre de secondes"""
        if isinstance(other, Temps):
            return self.ajouter_temps(other)
        else:
            return self.incremente(other)

    def ajouter_temps(self, other):
        """Addition de deux temps"""
        secondes = self.temps_vers_int() + other.temps_vers_int()
        return int_vers_temps(secondes)

    def incremente(self, secondes):
        """Addition un temps et un nombre de secondes"""
        secondes += self.temps_vers_int()
        return int_vers_temps(secondes)
```

Si other est un objet de type Temps, `__add__` invoque `ajouter_temps()`. Sinon, il suppose que le paramètre est un nombre et invoque incremente. Cette opération s'appelle **résolution de méthode basée sur le type**, car elle choisit la méthode à employer sur la base du type des arguments.

In [240]:

```
debut = Temps(9, 45)
print(debut + 1337)
```

```
10:7:17
```

la méthode spéciale `__radd__()`, qui signifie right-side add, « additionner à droite », est invoquée quand un objet de type Temps apparaît du côté droit de l'opérateur `+`. Voici la définition :

In [241]:

```
class Temps:
    """Représente le moment de la journée.
    attributs : heure, minute, seconde
    """
    def __init__(self, heure = 0, minute = 0, seconde = 0):
        """Initialise un objet Temps

        heure: int
        minute: int
        seconde: int or float
        """
        self.heure = heure
        self.minute = minute
        self.seconde = seconde

    def __str__(self):
        """Retourne un string Temps."""
        return "{}:{}:{}".format(self.heure, self.minute, self.seconde)

    def temps_vers_int(self):
        """Compte les secondes depuis minuit"""
        minutes = self.heure * 60 + self.minute
        secondes = minutes * 60 + self.seconde
        return secondes

    def __add__(self, other):
        """Addition d'un temps et un temps ou nombre de secondes"""
        if isinstance(other, Temps):
            return self.ajouter_temps(other)
        else:
            return self.incremente(other)

    def ajouter_temps(self, other):
        """Addition de deux temps"""
        secondes = self.temps_vers_int() + other.temps_vers_int()
        return int_vers_temps(secondes)

    def incremente(self, secondes):
        """Addition un temps et un nombre de secondes"""
        secondes += self.temps_vers_int()
        return int_vers_temps(secondes)

    def __radd__(self, other):
        """Addition d'un nombre de secondes et d'un temps"""
        return self.__add__(other)
```

In [242]:

```
debut = Temps(9, 45)
print(1337 + debut)
```

```
10:7:17
```

Polymorphisme

La résolution de méthode basée sur le type est utile quand elle est nécessaire, mais elle n'est pas toujours

nécessaire. Souvent, vous pouvez l'éviter en écrivant des fonctions qui s'exécutent correctement pour des arguments de types différents.

Les fonctions qui acceptent plusieurs types sont dites **polymorphes**. Le polymorphisme peut faciliter la réutilisation du code. Par exemple, la fonction interne `sum()`, qui ajoute des éléments à une séquence, fonctionne tant que les éléments de la séquence supportent l'addition.

Comme les objets de type Temps fournissent une méthode add, ils peuvent être passés en argument à `sum()` :

In [243]:

```
t1 = Temps(7, 43)
t2 = Temps(7, 41)
t3 = Temps(7, 37)
total = sum([t1, t2, t3])
print(total)
```

```
23:1:0
```

Méthodes Python

Voici ici une liste des noms de méthodes Python ainsi que leur équivalent mathématique ou autre:
Source: *https://docs.python.org/fr/3.6/library/operator.html (https://docs.python.org/fr/3.6/library/operator.html)*

Voici les opérateurs mathématiques

- `a + b` est équivalent à `__add__(a, b)` (addition)
- `a += b` est équivalent à `__iadd__(a, b)` (addition)
- `a - b` est équivalent à `__sub__(a, b)` (soustraction)
- `a -= b` est équivalent à `__isub__(a, b)` (soustraction)
- `a * b` est équivalent à `__mul__(a, b)` (mulitplication)
- `a *= b` est équivalent à `__imul__(a, b)` (mulitplication)
- `a / b` est équivalent à `__truediv__(a, b)` (division)
- `a // b` est équivalent à `__floordiv__(a, b)` (division entière)
- `a % b` est équivalent à `__mod__(a, b)` (modulo)
- `a ** b` est équivalent à `__pow__(a, b)` (exponentiation)

Voici les opérateurs de comparaison

- `a < b` est équivalent à `__lt__(a, b)` (less then)
- `a <= b` est équivalent à `__le__(a, b)` (less or equal)
- `a == b` est équivalent à `__eq__(a, b)` (equal)
- `a != b` est équivalent à `__ne__(a, b)` (not equal)
- `a >= b` est équivalent à `__ge__(a, b)` (greater or equal)
- `a > b` est équivalent à `__gt__(a, b)` (greater)

Exemples:

In [244]:

```
class Test:
    """Classe cree afin de tester la surcharge d'operateur"""

    def __init__(self, a, b, c):
        """Initialise un objet Test"""
        self.a = a
        self.b = b
        self.c = c

    def __str__(self):
        """Retourne l'objet Test"""
        return "{},{},{}".format(self.a, self.b, self.c)

    def __mul__(self, other):
        """Mutiplie deux objets Test"""
        return Test(self.a, self.b*other.b, other.c)

    def __eq__(self, other):
        """Retourne True si les attributs b sont egaux"""
        if self.b == other.b:
            return True
        return False
```

In [245]:

```
Test1 = Test(1, 2, 1)
Test2 = Test(2, 2, 2)

Test3 = Test1 * Test2
print(Test3)
```

```
1,4,2
```

In [246]:

```
Test1 == Test2
```

Out[246]:

```
True
```

Héritage

La fonctionnalité la plus emblématique de la programmation orientée objet est l'**héritage**. L'héritage est la possibilité de définir une nouvelle classe, qui est une version modifiée d'une classe existante.

Si nous voulons définir un nouvel objet pour représenter une carte à jouer, il est évident que les attributs doivent être la couleur et la valeur. Le type des attributs n'est pas si évident. Une possibilité est d'utiliser des chaînes contenant des mots comme 'pique' pour les couleurs et 'dame' pour les valeurs. Un problème avec cette modélisation est qu'il ne serait pas facile de comparer les cartes pour voir laquelle a une valeur ou une couleur supérieure.

Une autre possibilité est d'utiliser des entiers pour **encoder** les valeurs et les couleurs. Dans ce contexte, « encoder » signifie que nous allons définir une correspondance entre nombres et couleurs, ou entre nombres et valeurs.

In [247]:

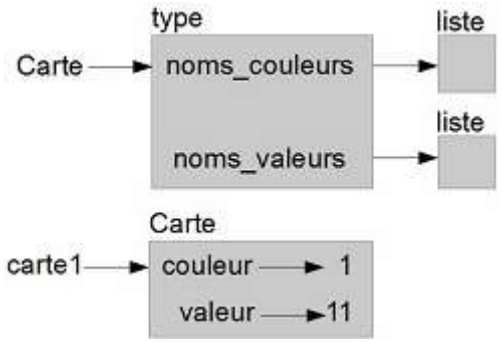
```
class Carte:
    """Représente une carte à jouer standard."""
    def __init__(self, couleur = 0, valeur = 2):
        self.couleur = couleur
        self.valeur = valeur

    noms_couleurs = ['trèfle', 'carreau', 'coeur', 'pique']
    noms_valeurs = [None, 'as', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'valet', 'dam

    def __str__(self):
        """Retourne le nom d'une carte"""
        return "{} de {}".format(Carte.noms_valeurs[self.valeur], Carte.noms_couleurs[self.
```

Les variables comme noms_couleurs et noms_valeurs, qui sont définies dans une classe, mais en dehors de toute méthode, s'appellent **attributs de classe** parce qu'elles sont associées à l'objet classe Carte.
Ce terme les distingue des variables telles que couleur et valeur, qui s'appellent **attributs d'instance** parce qu'elles sont associées à une instance particulière.

In [248]:

```
carte1 = Carte(1, 11)
print(carte1)
```

```
valet de carreau
```



Maintenant que nous avons les cartes, la prochaine étape est de définir les Paquets de cartes. Comme un paquet est composé de cartes, il est naturel que chaque Paquet contienne comme attribut une liste de cartes. Ce qui suit est une définition de classe pour Paquet. La méthode `__init__()` crée l'attribut cartes et génère l'ensemble standard de cinquante-deux cartes :

In [249]:

```
class Paquet:
    """Represente un paquet de 52 cartes"""
    def __init__(self):
        self.cartes = []
        for couleur in range(4):
            for valeur in range(1, 14):
                carte = Carte(couleur, valeur)
                self.cartes.append(carte)

    def __str__(self):
        """Retourne toutes les cartes du paquet"""
        res = []
        for carte in self.cartes:
            res.append(str(carte))
        return ", ".join(res)

    def pop_carte(self):
        """Enleve une carrte du paquet"""
        return self.cartes.pop()

    def ajouter_carte(self, carte):
        """Ajoute une carte au paquet"""
        self.cartes.append(carte)

    def battre(self):
        """Melange le paquet"""
        random.shuffle(self.cartes)
```

Une méthode comme `ajouter_carte()` , qui utilise une autre méthode sans faire beaucoup de travail s'appelle parfois un **placage**.
Dans ce cas, `ajouter_carte()` est une méthode « mince » qui exprime une opération de liste en termes appropriés pour les paquets. Elle améliore l'apparence, ou l'interface, de la mise en oeuvre.

In [250]:

```
paquet = Paquet()
print(paquet)
```

```
as de trèfle, 2 de trèfle, 3 de trèfle, 4 de trèfle, 5 de trèfle, 6 de trèfl
e, 7 de trèfle, 8 de trèfle, 9 de trèfle, 10 de trèfle, valet de trèfle, dam
e de trèfle, roi de trèfle, as de carreau, 2 de carreau, 3 de carreau, 4 de
carreau, 5 de carreau, 6 de carreau, 7 de carreau, 8 de carreau, 9 de carrea
u, 10 de carreau, valet de carreau, dame de carreau, roi de carreau, as de c
oeur, 2 de coeur, 3 de coeur, 4 de coeur, 5 de coeur, 6 de coeur, 7 de coeu
r, 8 de coeur, 9 de coeur, 10 de coeur, valet de coeur, dame de coeur, roi d
e coeur, as de pique, 2 de pique, 3 de pique, 4 de pique, 5 de pique, 6 de p
ique, 7 de pique, 8 de pique, 9 de pique, 10 de pique, valet de pique, dame
de pique, roi de pique
```

L'**héritage** est la capacité de définir une nouvelle classe qui est une version modifiée d'une classe existante. À titre d'exemple, disons que nous voulons une classe pour représenter une « main », c'est-à-dire les cartes détenues par un seul joueur. Une main est semblable à un paquet : les deux sont constitués d'une collection de cartes, et les deux nécessitent des opérations comme l'ajout et le retrait de cartes.
En même temps, une main est différente d'un paquet ; il existe des opérations que nous voulons pour les «

mains » qui n'ont pas de sens pour un paquet. Par exemple, au poker, nous pourrions comparer deux mains pour voir qui gagne. Au bridge, nous pourrions calculer le nombre de points d'une main afin de faire une enchère.

In [251]:

```
class Main(Paquet):
    """Représente une main au jeu de cartes."""
```

Lorsqu'une nouvelle classe hérite d'une classe existante, la classe existante est appelée **classe mère** ou **classe parente** et la nouvelle classe est appelée **classe fille** ou **classe enfant**.

In [252]:

```
class Main(Paquet):
    """Représente une main au jeu de cartes."""

    def __init__(self, etiquette = ''):
        """Retourne toutes les cartes de la main"""
        self.cartes = []
        self.etiquette = etiquette
```

Si nous fournissons une méthode d'initialisation à la classe Main, elle remplace celle de la classe Paquet. Lorsque vous créez une Main, Python appelle cette méthode `__init__()` , pas celle de Paquet.

In [253]:

```
main = Main('nouvelle main')
main.etiquette
```

Out[253]:

```
'nouvelle main'
```

In [254]:

```
paquet = Paquet()
carte = paquet.pop_carte()
main.ajouter_carte(carte)
print(main)
```

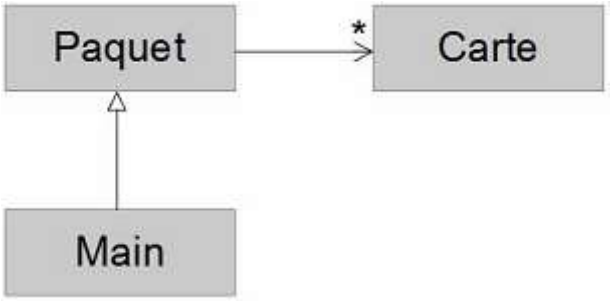
```
roi de pique
```

In [255]:

```
class Paquet:
    """Represente un paquet de 52 cartes"""
    def __init__(self):
        self.cartes = []
        for couleur in range(4):
            for valeur in range(1, 14):
                carte = Carte(couleur, valeur)
                self.cartes.append(carte)

    def __str__(self):
        """Retourne toutes les cartes du paquet"""
        res = []
        for carte in self.cartes:
            res.append(str(carte))
        return ", ".join(res)

    def pop_carte(self):
        """Enleve une carrte du paquet"""
        return self.cartes.pop()

    def ajouter_carte(self, carte):
        """Ajoute une carte au paquet"""
        self.cartes.append(carte)

    def battre(self):
        """Melange le paquet"""
        random.shuffle(self.cartes)

    def deplacer_cartes(self, main, nombre):
        """Deplace n cartes d'un paquet a un autre"""
        for i in range(nombre):
            main.ajouter_carte(self.pop_carte())
```

Il existe plusieurs types de relations entre les classes :

- les objets d'une classe peuvent contenir des références vers des objets d'une autre classe. Par exemple, chaque Rectangle contient une référence vers un Point, et chaque Paquet contient des références vers plusieurs Cartes. Ce type de relation est appelé **HAS-A**, « a-un(e) », comme dans « un Rectangle a un Point » ;
- une classe peut hériter d'une autre. Cette relation est appelée **IS-A**, « est-un(e) », comme dans « une Main est une sorte de Paquet. » ;
- une classe peut dépendre d'une autre dans le sens où les objets d'une classe prennent comme paramètres des objets de la seconde classe, ou utilisent des objets de la seconde classe dans le cadre d'un calcul. Ce type de relation est appelée une **dépendance**.



Ceci est un **diagramme de classes**

La flèche à pointe triangulaire creuse représente une relation IS-A ; dans ce cas, elle indique que Main hérite de Paquet.
La flèche à pointe normale représente une relation HAS-A ; dans ce cas, un Paquet a des références vers des objets Carte.

L'astérisque près de la pointe de la flèche est une **multiplicité** ou **cardinalité** ; il indique combien de Cartes a un Paquet. Une multiplicité peut être un simple nombre, comme 52, une plage de valeurs, comme 5..7 ou une étoile, qui indique qu'un Paquet peut avoir un nombre quelconque de Cartes.

Une dépendence est normalement représentées par une flèche en pointillé. Ou s'il y a beaucoup de dépendances, elles sont parfois omises.

Parfois, il est moins évident de déterminer quels sont les objets dont vous avez besoin et comment ils doivent interagir. Dans ce cas, vous avez besoin d'un modèle de développement différent. De la même manière que nous avons découvert des interfaces de fonction par encapsulation et généralisation, nous pouvons découvrir des interfaces de classe par **encapsulation de données**. C'est à dire un modèle de développement d'un programme qui implique au départ un prototype utilisant des variables globales et une version finale qui transforme les variables globales en attributs d'instance.