Programmation orientée objet avec Python

Marie-Dominique Van Damme, ENSG

Cycle d'ingénieur - Master mention géomatique
mars 2024

Source de la section d'introduction issue du cours : Programmation orientée objet avec python, M-D. Van Damme, Y. Méneroux, 2020-2021

Table des matières

Différents styles de programmation

- **Impératif** (C, Python) : séquences d'instructions indiquant comment on obtient un résultat en manipulant la mémoire
- **Déclaratif** (Prolog, SQL) : décrit ce que l'on a, ce que l'on veut, et non pas comment on l'obtient,
- **Fonctionnel** (Lisp, Java, Python) : évaluation d'expressions/fonctions où le résultat ne dépend pas d'un état interne,
- **Objet** (C++, Java, Python) : ensembles d'objets qui possèdent un état interne et des méthodes qui interrogent ou modifient cet état.

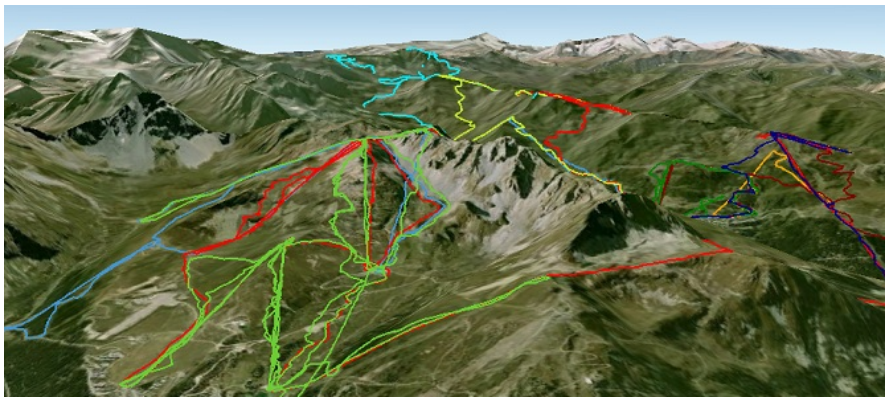
Plan

Plan

Exemple d'introduction

Problème : on voudrait écrire un programme pour faire quelques interrogations et des statistiques de qualité sur un jeu de traces de données GPS. Par exemple :

- Quelles sont les traces qui ont une longueur supérieure à 100 km ?
- Quelles sont les traces qui passent non loin du refuge XXX (Lon, Lat) ?
- Les vitesses entre 2 points sont-elles conformes aux attentes ?



Source de la section d'introduction issue du cours : Programmation orientée objet avec python, M-D. Van Damme, Y. Méneroux, 2020-2021

Exemple d'introduction

Quelle structure de données pour stocker les informations ?

Point ?

Exemple d'introduction

Quelle structure de données pour stocker les informations ?

Point : liste de taille 3 (lon, lat, timestamp)
$P = [x_1, x_2, x_3]$

Exemple d'introduction

Quelle structure de données pour stocker les informations ?

Point : liste de taille 3 (lon, lat, timestamp)
$P = [x_1, x_2, x_3]$

Trace ? liste de liste de taille 3
$T = [[x_{11}, x_{12}, x_{13}], [x_{21}, x_{22}, x_{23}], ..., [x_{N1}, x_{N2}, x_{N3}]]$

Exemple d'introduction

Quelle structure de données pour stocker les informations ?

Point : liste de taille 3 (lon, lat, timestamp)

$P = [x_1, x_2, x_3]$

Trace : liste de liste de taille 3

$T = [[x_{11}, x_{12}, x_{13}], [x_{21}, x_{22}, x_{23}], ..., [x_{n1}, x_{n2}, x_{n3}]]$

Dataset ? liste de liste de liste de taille 3

$$D = [[[x^1_{11}, x^1_{12}, x_{13}], [x^1_{21}, x^1_{22}, x^1_{23}], ..., [x^1_{n_1 1}, x^1_{n_1 2}, x^1_{n_1 3}]],$$
$$[[[x^p_{11}, x^p_{12}, x^p_{13}], [x^p_{21}, x^p_{22}, x^p_{23}], ..., [x^p_{n_m 1}, x^p_{n_m 2}, x^p_{n_m 3}]]]$$

Exemple d'introduction

Quelle structure de données pour stocker les informations ?

Point : liste de taille 3 (lon, lat, timestamp)
$P = [x_1, x_2, x_3]$

Trace : liste de liste de taille 3
$T = [[x_{11}, x_{12}, x_{13}], [x_{21}, x_{22}, x_{23}], ..., [x_{N1}, x_{N2}, x_{N3}]]$

Dataset : liste de de liste de liste de taille 3
$D = [[[x^1_{11}, x^1_{12}, x_{13}], [x^1_{21}, x^1_{22}, x^1_{23}], ..., [x^1_{n_1 1}, x^1_{n_1 2}, x^1_{n_1 3}]],$
$[[[x^p_{11}, x^p_{12}, x^p_{13}], [x^p_{21}, x^p_{22}, x^p_{23}], ..., [x^p_{n_m 1}, x^p_{n_m 2}, x^p_{n_m 3}]]]$

x^k_{ij} : i^{eme} coordonnée du j^{eme} point
de la trace du randonneur k

Exemple d'introduction

Quelle structure de données pour stocker les informations ?

liste de liste de liste de taille 3

x_{ij}^{k} : i^{eme} coordonnée du j^{eme} point
de la trace du randonneur k

En Python, on prendrait une structure *list* : TAB[k][i][j]
$\Rightarrow$ pas facile à manipuler. Pour preuve :

Exemple d'introduction

Quelles sont les traces qui ont une distance supérieure à 100 km ?

```
for numTrace in range(len(dataset)):
    distance = 0
    for i in range(len(dataset[numTrace])):
        if i > 0:
            pointD = dataset[numTrace][i-1]
            pointF = dataset[numTrace][i]
            distance += math.sqrt(
                            (pointF[0] - pointD[0])**2
                          + (pointF[1] - pointD[1])**2)
        if distance > 100:
            print ('Trace %d depasse 100km' %numTrace)
```

Exemple d'introduction

Quelles sont les traces qui passent non loin (à moins de) du refuge XXX ?

```
xRefuge, yRefuge = 5, 5
distProche, traceProche = 3, list()
for numTrace in range(len(dataset)):
    for i in range(len(dataset[numTrace])):
        point = dataset[numTrace][i]
        ecartPointRefuge = math.sqrt(
                  (point[0] - xRefuge)**2
                + (point[1] - yRefuge)**2)
        if ecartPointRefuge < distProche:
            try:
                pos = traceProche.index(numTrace)
            except ValueError:
                traceProche.append(numTrace)
print ("Traces passant a cote du refuge: ", traceProche)
```

Exemple d'introduction

Nouvelle demande, on voudrait étudier la qualité sur ces traces. Pour cela on va contrôler la vitesse entre 2 points. Quel impact sur la structure définie précédemment ?

=> il faut casser la structure pour introduire une structure intermédiaire "tronçon" (une "4 ème liste")

=> re-programmer les 2 fonctions précédentes

Exemple d'introduction : conclusion

Comment programmer simplement des actions simples sur des éléments variés et complexes ?

Comment ajouter des fonctions sans tout réécrire et tout retester ?

Les fonctions permettent de factoriser les traitements qui agissent sur les structures de données, est-ce qu'il existe quelque chose qui puisse définir ses propres types de données ?

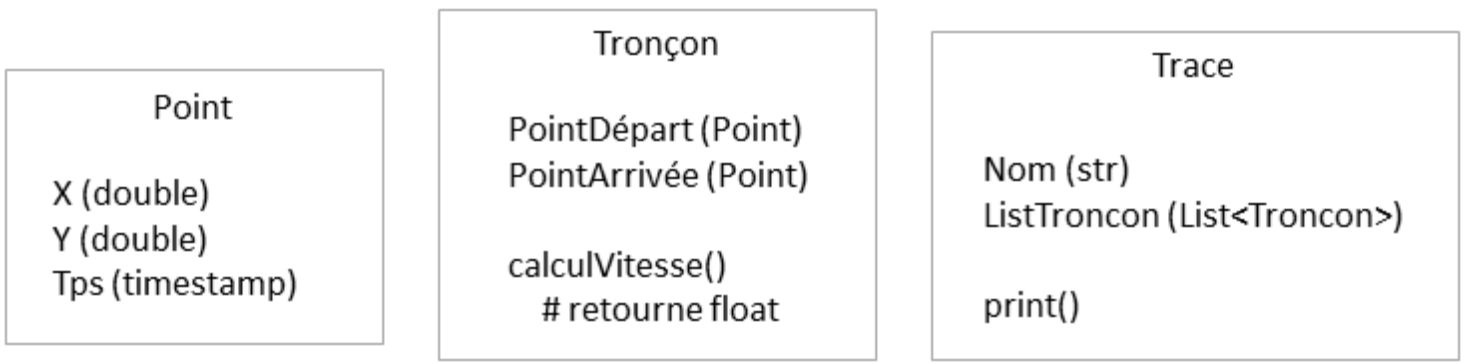


Figure – Types de structures envisagés

Exemple d'introduction : conclusion

La programmation orientée objet va nous permettre de manipuler des types de données plus complexes :

Récupération de la trace k

```
trace = dataset.getTrace(k)
```

Exemple d'introduction : conclusion

La programmation orientée objet va nous permettre de manipuler des types de données plus complexes :

Récupération du point *j* de la trace *k*

```
point = dataset.getTrace(k).getPoint(j)
```

Exemple d'introduction : conclusion

La programmation orientée objet va nous permettre de manipuler des types de données plus complexes :

Récupération du timestamp ($j = 3$) du point j de la trace k

```
tps =
dataset.getTrace(k).getPoint(j).getTimestamp()
```

Exemple d'introduction : conclusion

La programmation orientée objet nous permettra également d'empaqueter un certain nombre de méthodes avec les datasets de points GPS :

Calcul du temps de parcours de la 1ere trace

```
trace = dataset.getTrace(1)
temps_de_parcours =
trace.calculTempsParcours()
```

Calcul de la vitesse max de la 10e trace

```
trace = dataset.getTrace(10)
temps_de_parcours = trace.calculMaxSpeed()
```

Calcul du nombre de traces

```
nb_traces = dataset.getNumberOfTraces()
```

Plan

POO - Introduction

La clé pour bien comprendre la **programmation orientée objet** c'est de voir les objets comme des collections mélangeant des données et des fonctions qui agissent sur ces données.

POO - Introduction

- On a déjà vu que tout objet que python manipulait, avait un **type**. C'est le type de l'objet qui définissait les opérations que l'on pouvait faire avec.

- On s'est appuyé sur des objets de type structuré : les listes (list), les chaines de caractères (strings), les dictionnaires (dicts).
  Ces types structurés étaient dotés d'une série de fonctions avec lesquelles on pouvait manipuler ces structures de données : append, split, etc.

Introduction

On va maintenant voir un mécanisme qui permet de définir de nouveaux types.

La programmation orientée objet, c'est un style de programmation qui permet de regrouper au même endroit le comportement (les fonctions) et les données (les structures) qui sont faites pour aller ensemble.

C'est une simple question d'organisation du programme.

Plusieurs langages supportent l'orienté objet ou assimilé : Python, Java, C++, Javscript, PHP, Ada, Visual Basic...

Introduction

Le paradigme de la programmation orientée objet n'est donc jamais indispensable, ni plus optimisé (en terme de temps de calcul).

Mais il est plus simple à utiliser (travail en équipe), à maintenir et correspond bien souvent à une expression plus naturelle du code vis-à-vis du problème à modéliser.

Plan

Types de données abstraits

Types primitifs :

- Booléens : `True` ou `False`
- Entiers : 2, 3, -10, 6.02×10^{23}...
- Flottants : 9.81, -154.36, 3.14151, 6.67×10^{-11}...

Types de données abstraits

La programmation orientée objet nous donne les moyens de créer à volonté de nouveaux objets et d'empaqueter dans la même structure, **l'ensemble des données et des fonctions** nécessaires à la manipulation de l'objet.

Une **classe** est un nouveau type de données. Une classe est la description formelle d'un ensemble d'objets ayant une sémantique et des propriétés communes.

Types de données abstraits

- Les attributs sont des entités qui définissent les propriétés d'objets. Par exemple un point GPS est défini par des coordonnées spatiales (type *float*) dans un système de référence (type *?*) et par un timestamp (type *datetime*).

- Les méthodes permettent de définir les actions que peuvent réaliser ces objets. Par exemple comment définir la méthode qui transforme un point GPS d'un système de référence dans un autre ?

Types de données abstraits

En Python, certains types d'objets sont déjà pré-définis dans le langage comme par exemple :

Les listes : `L.sort()`, `L.append()`...
Les chaînes de caractères : `c.lowercase()`, `c.split(",")`...

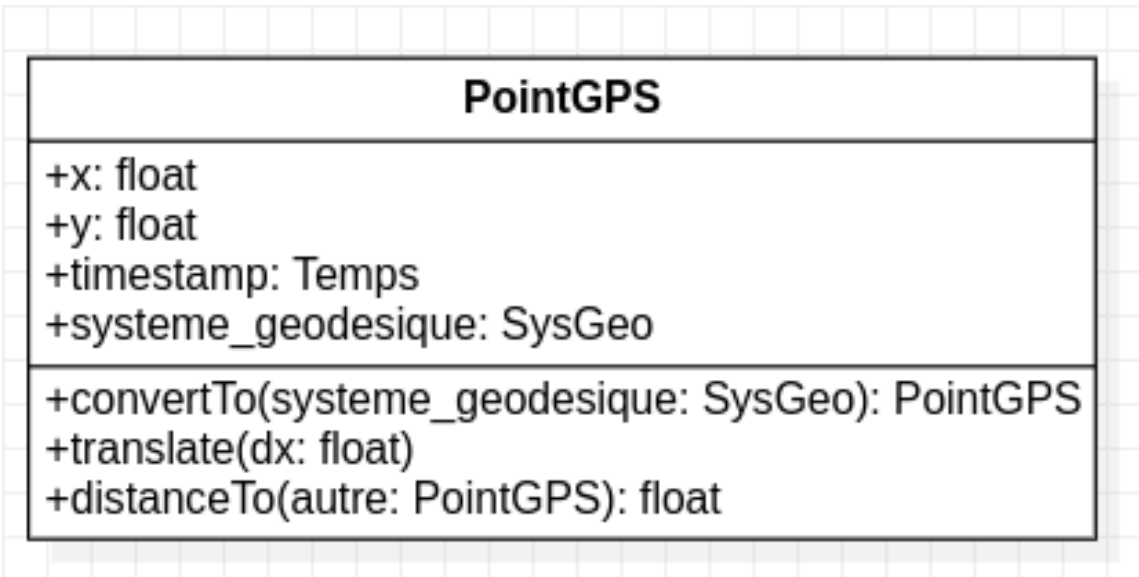
L'existence de deux méthodes de tri sur les listes exprime la dualité entre les deux méthodes de programmation :

`L = sorted(L)` → programmation classique
`L.sort()` → programmation orientée objet

Types de données abstraits

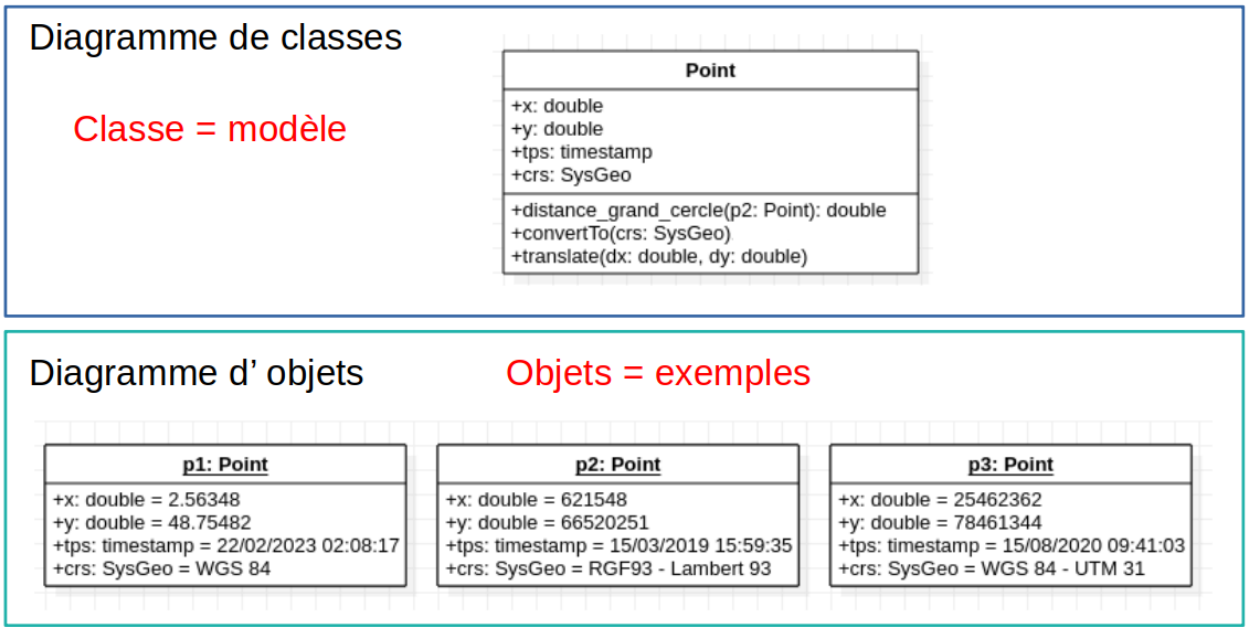
Une classe est représentée par un rectangle divisé en trois compartiments. Le premier indique le nom de la classe, le deuxième ses attributs (les données structurées) et le troisième ses méthodes (les fonctions).

Par exemple une classe PointGPS peut avoir comme représentation graphique :



Types de données abstraits

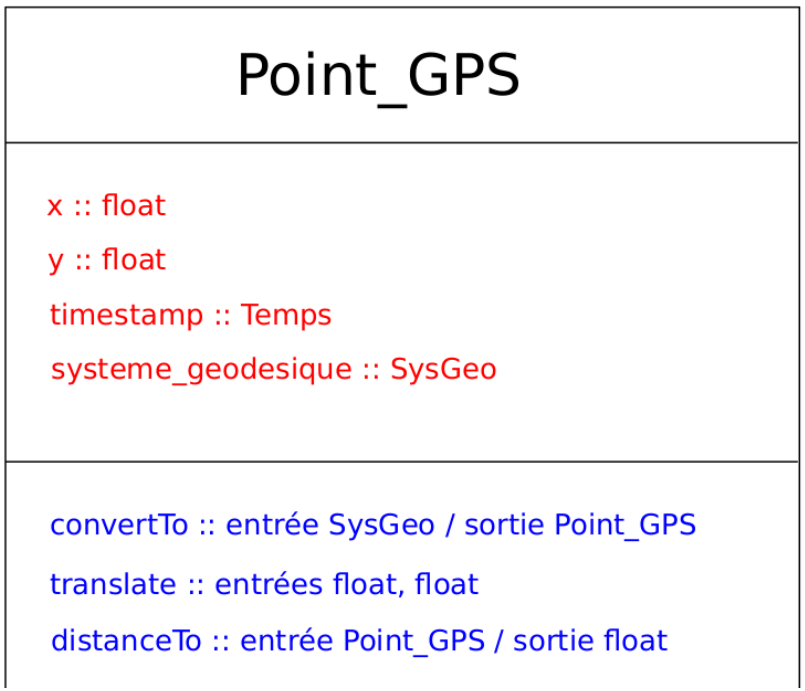
Un **objet** est une instance d'une classe. C'est une entité dotée d'une identité, d'un état et de comportements que l'on peut appeler.



Les objets sont des éléments individuels d'un système en cours d'exécution.

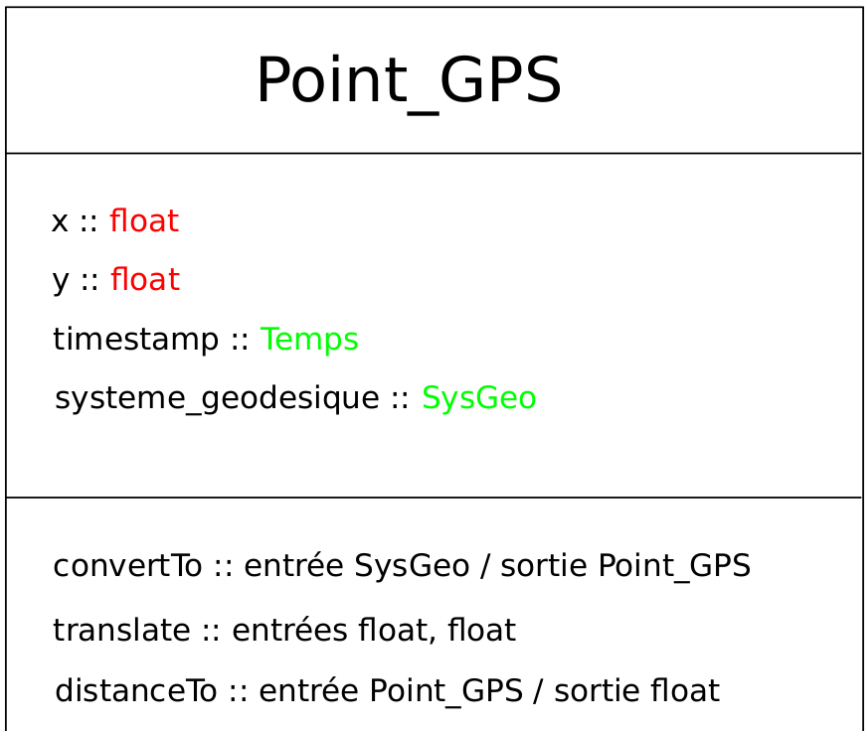
Types de données abstraits

Une classe contient des attributs et des méthodes permettant de manipuler facilement ces données :



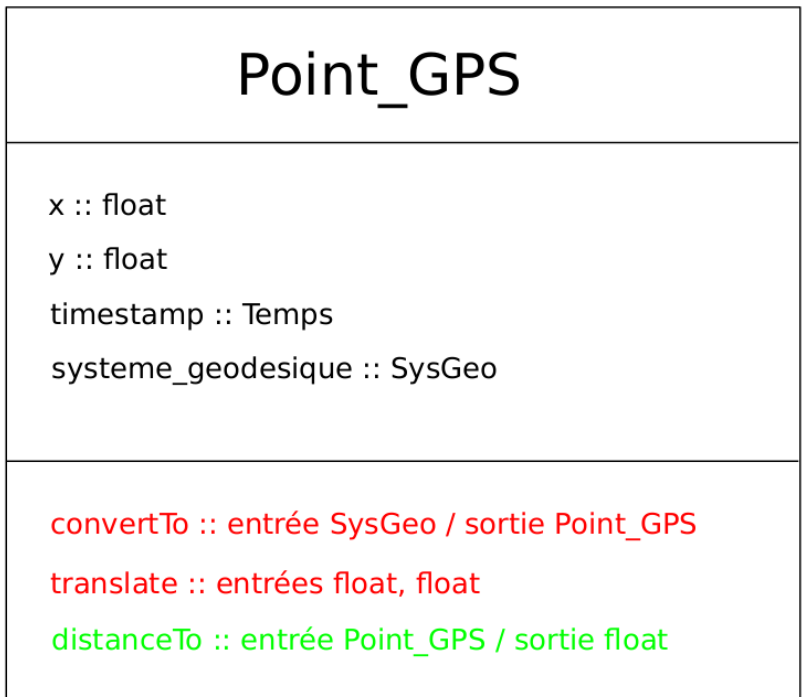
Types de données abstraits

Une classe peut contenir des attributs de type primitif ou/et des attributs de type objet



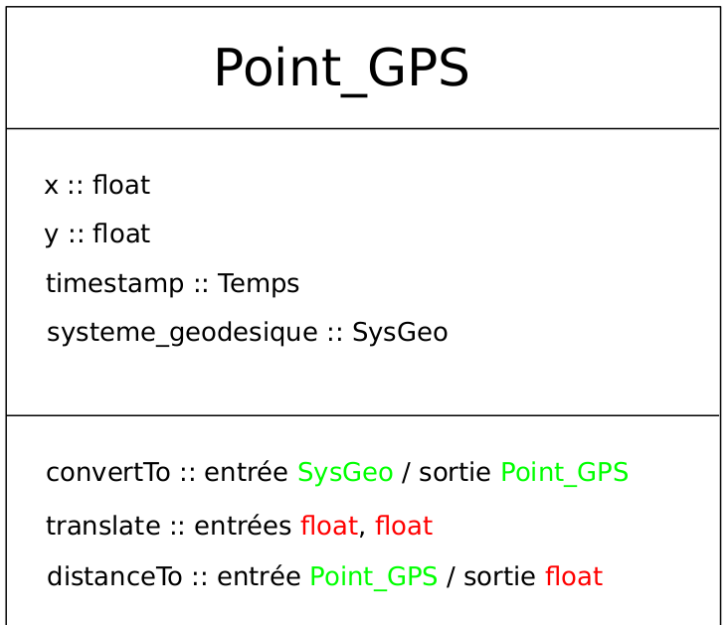
Types de données abstraits

Les méthodes d'une classe peuvent agir sur l'objet lui même (c.à.d modifier ses attributs) ou retourner une information sur l'objet (calculée à partir de ses attributs)



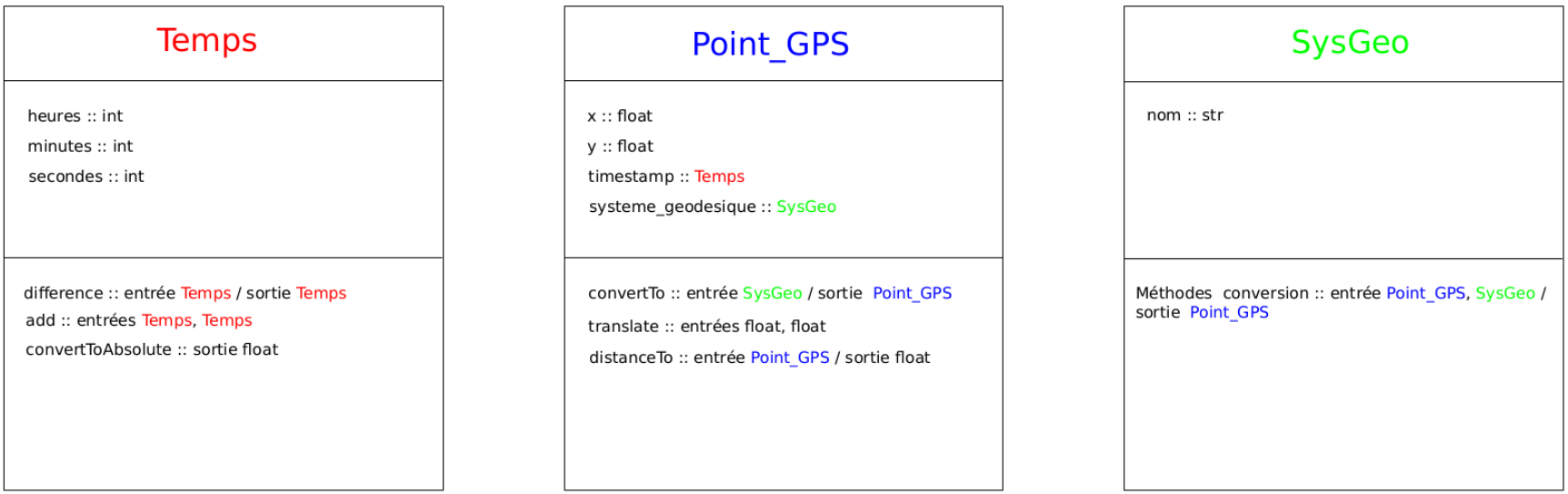
Types de données abstraits

Les méthodes d'une classe peuvent prendre en entrées et en sortie, des types primitifs ou des objets



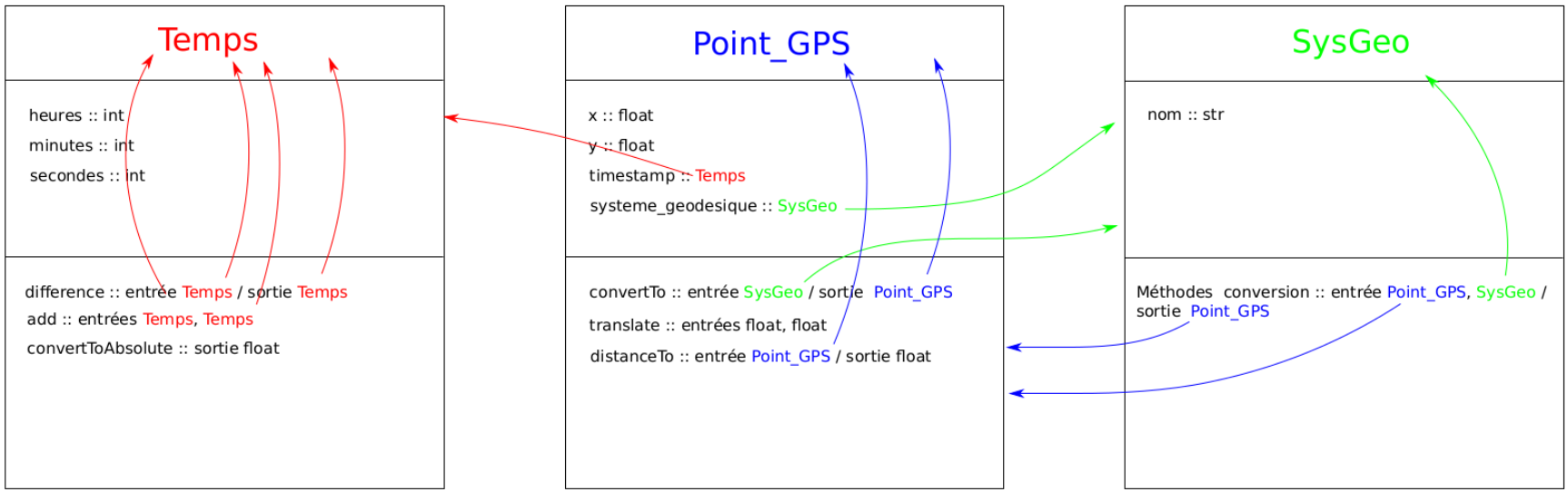
Types de données abstraits

Bien entendu, tous les types d'objets utilisés dans une classe devront avoir été définis préalablement.



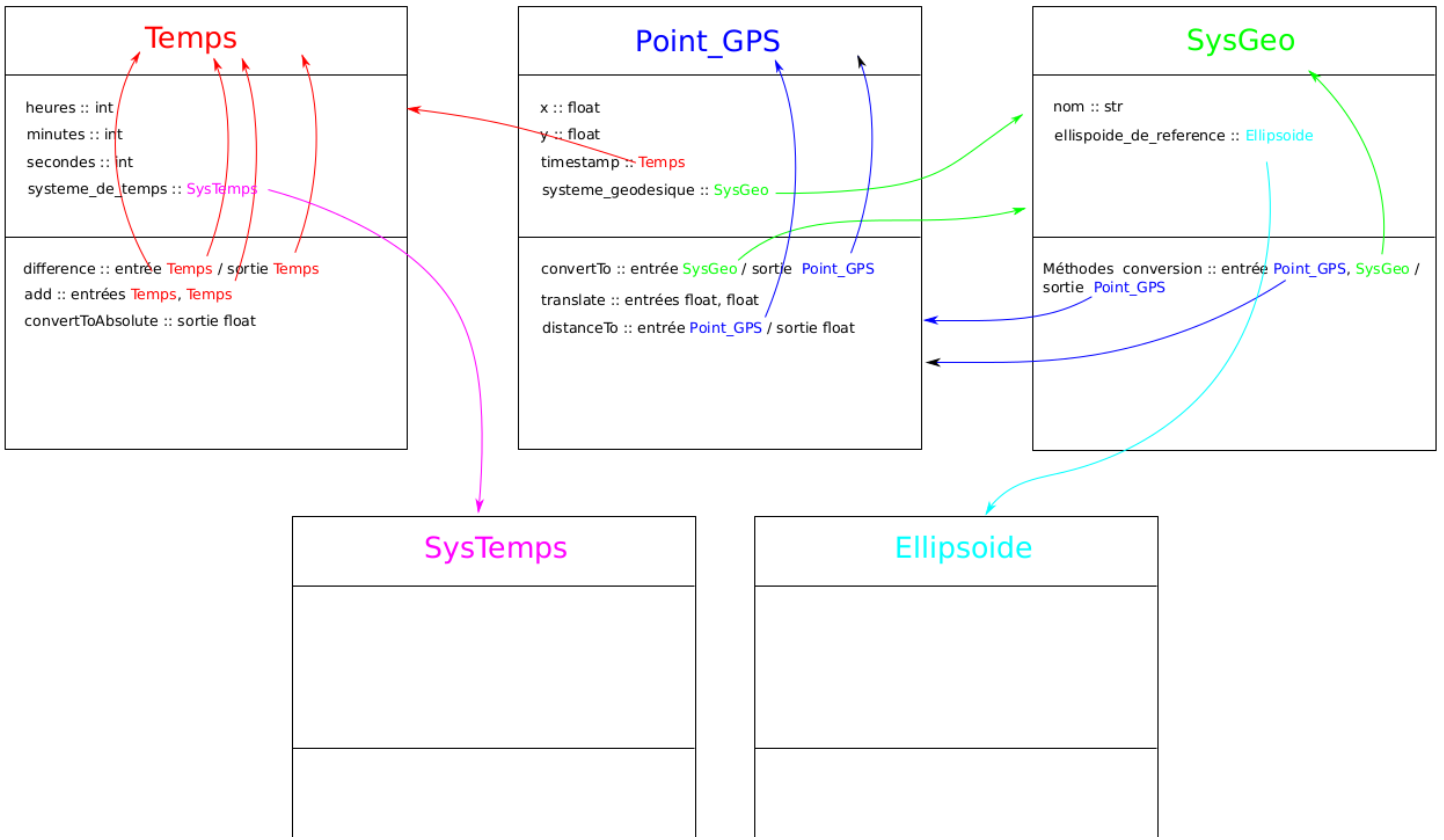
Types de données abstraits

Bien entendu, tous les types d'objets utilisés dans une classe devront avoir été définis préalablement. L'ensemble constitue un framework orienté objet.



Types de données abstraits

Bien entendu, tous les types d'objets utilisés dans une classe devront avoir été définis préalablement. Le framework est complexifiable à volonté, en fonction des besoins.



Types de données abstraits

⚠ Deux opérations spécifiques aux objets :

- La construction (ou instanciation) d'un objet
  `p1 = Point(56.3, 42.7)` (en python)
  `Point p1 = new Point(56.3, 42.7)` (en Java)

- L'accès **par référence** aux attributs et aux méthodes avec l'utilisation du "."
  `coord_x = p1.x` (en python et en Java)
  `dist = p1.distanceTo(p2)` (en python et en Java)

Interfaces

Les **spécifications** des opérations d'un type abstrait définissent une **interface** entre le type de données abstrait et le programme.

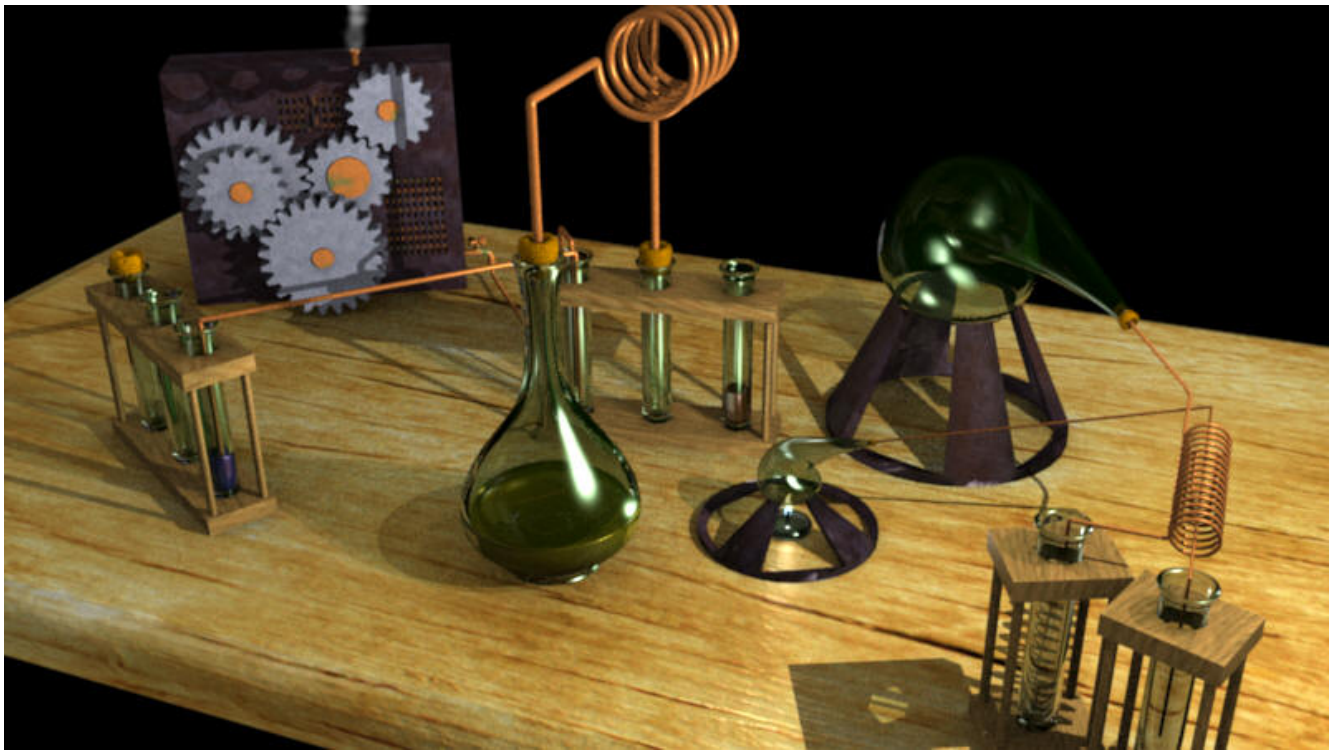
L'interface définit le comportement des opérations : ce qu'elles font, mais sans dire comment elles le font.

L'interface fournit ainsi une barrière d'abstraction qui isole du reste du programme, par les structures de données, les algorithmes fournissant l'implémentation des types abstraits.

L'**Encapsulation** permet de protéger l'information contenue dans un objet et son fonctionnement interne, et de le rendre manipulable uniquement par certaines de ses méthodes. Les détails de l'implémentation sont donc masqués à l'utilisateur.

Interfaces

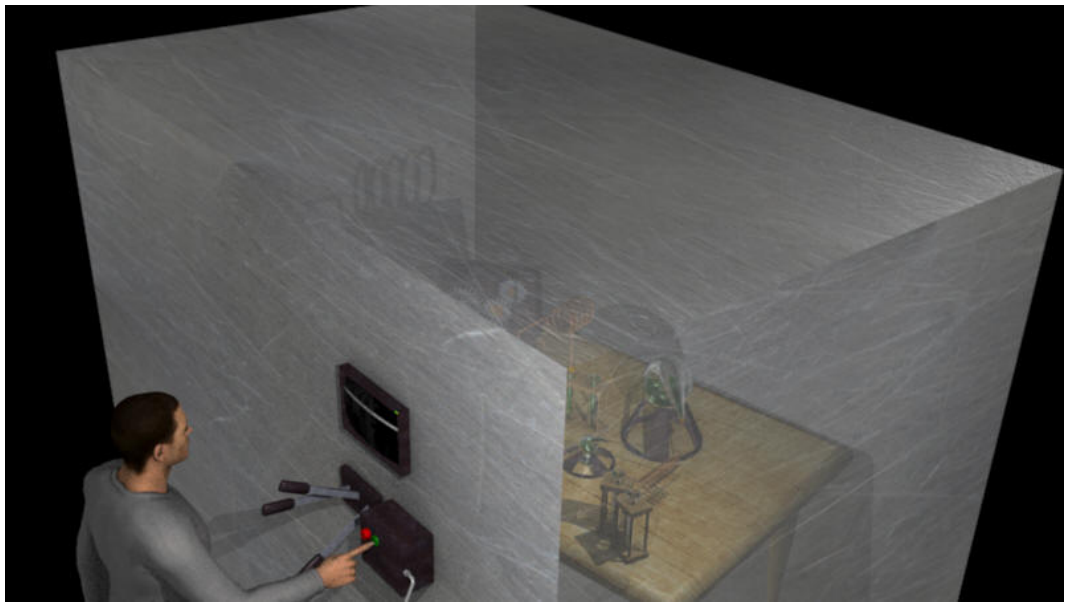
Quand on code un programme, on ajoute beaucoup de variables et de fonctions qui s'entre-appellent :



Référence : Mathieu Nebra, Open classroom

Interfaces

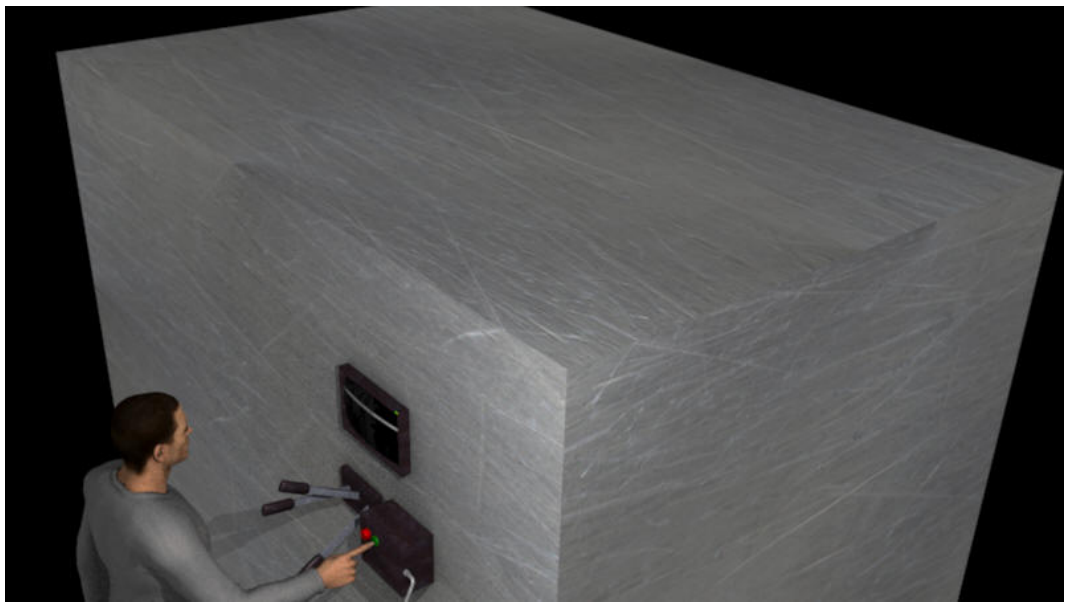
En programmation orientée objet, le code complexe peut-être encapsulé dans des classes qui seront opaques pour l'utilisateur => la complexité est masquée.



Référence : Mathieu Nebra, Open classroom

Interfaces

Seules quelques méthodes des classes sont accessibles :



Référence : Mathieu Nebra, Open classroom

Plan

Plan

Classe

Une **classe** est un "moule" à objets. Elle définit un nouveau type de données. Ci-dessous une définition d'une classe écrite dans le langage *Python* qui fournit une implémentation des coordonnées d'un point dans un espace vectoriel à 2 dimensions :

```
class Coord:
    """
    Classe pour représenter les coordonnées d'un point
          dans un espace vectoriel à 2 dimensions.
    """
    pass
```

En python une classe est définie de manière analogue aux fonctions mais en utilisant le mot clé *class*

Nommage

Par convention, on nomme les classes par une première lettre en majuscule et les différents mots sont également indiqués par une première en majuscule et sont sans espace. Cette convention suit le principe de l'écriture en dromadaire (camel case) : *MaClasse*.

On a par exemple : *Curve*, *LineString*, *MultiLineString*, *SmallestSurroundingRectangle*, etc.

Instance

Pour créer un objet, ici une coordonnée, il suffit d'appeler la classe *Coord* :

```
c1 = Coord()
```

La variable *c1* référence l'objet crée. On dit que l'objet est une instance de *Coord*. Une classe définit un nouveau type dans le langage. La POO est d'abord un modèle de programmation qui permet d'ajouter dans le langage des nouveaux types d'objet.

Connaître un type abstrait

En Python, il existe deux fonctions standards intéressantes.

- 1/ La fonction ***type(o)*** retourne le type de l'objet passé en paramètre :

```
type([1,2,3])
type(c1)
```

```
??
??
```

Connaître un type abstrait

En Python, il existe deux fonctions standards intéressantes.

- 1/ La fonction ***type(o)*** retourne le type de l'objet passé en paramètre :

```
type([1,2,3])
type(c1)
```

```
list
<class '__main__.Coord'>
```

Tester un type abstrait

- 2/ La fonction ***isinstance(o, cls)*** retourne *True* si l'objet passé en premier paramètre est une instance de la classe passée en deuxième paramètre :

```
isinstance(c1, Coord)
isinstance(c1, list)
isinstance(c1, str)
```

```
??
??
??
```

Tester un type abstrait

- 2/ La fonction ***isinstance(o, cls)*** retourne *True* si l'objet passé en premier paramètre est une instance de la classe passée en deuxième paramètre :

```
isinstance(c1, Coord)
isinstance(c1, list)
isinstance(c1, str)
```

```
True
False
False
```

Plan

Les attributs

Définition

Les données d'une classe sont stockées dans des variables appelées **attributs**.

Dans le langage python, un attribut est un nom écrit juste après un point.

On peut créer un attribut, changer sa valeur depuis l'extérieur du corps de la classe :

```
c1 = Coord()
c1.x = 3
c1.y = 5
```

Les attributs

Il est recommandé de donner aux attributs des noms écrits en minuscules : couleur, nom, code, etc.

On utilise un double underscore devant son nom pour signifier qu'il s'agit d'un attribut privé

Classe

Quand on définit une fonction dans une définition de classe, la définition de la fonction s'appelle **méthode**.

La classe déclare des **attributs** représentant l'état des objets et les **méthodes** représentant alors leur comportement.

Méthodes

Une méthode est une fonction dans la classe et utilise donc le mot clé "def"

```
class Coord:
    def f(......):
        # ...

    def g(....):
        # ....
```

Plan

Instancier une classe

Définition

Instancier une classe *A* signifie créer un objet de type *A*

L'instanciation est une opération simple avec le langage Python, il suffit d'appeler la classe :

```
obj = A()
```

```
L = list("abcd")
print (L)
```

```
['a', 'b', 'c', 'd']
```

Instancier une classe

Lors d'un appel du type : *c2 = Coord(x, y)*, le mécanisme est le suivant :

- Python lance automatiquement et de manière transparente le constructeur de *Coord* en lui faisant passer les arguments *Coord*, *x* et *y*. Le constructeur retourne <instance> de type *Coord* ;
- Python lance, toujours de manière transparente, l'initialiseur de *Coord* en lui faisant passer les arguments <instance>, x et y. Cet appel équivaut à ceci :

```
Coord.__init__(<instance>, x, y)
```

  L'initialiseur a vocation à initialiser des attributs de <instance> ;
- Enfin, Python fait pointer le nom *c2* vers <instance>

Instancier une classe

La méthode __init__ est une méthode commune à toutes les classes. On peut bien sûr la personnaliser afin de les adapter aux besoins :

```
class Coord:
    def __init__(self, x, y):
        """
        Creation d'une nouvelle coord x, y.
        """
        self.x = x
        self.y = y
```

Cette méthode permet d'initialiser les attributs x et y d'une instance.

Self

Le premier argument d'une méthode doit être **self**, argument obligatoire. Cet objet self est une référence à l'instance.

Quand un objet fait référence à une méthode, le paramètre "self" est implicitement passé, on ne doit pas le préciser :

```
>>> c0.translate(0.25, 1.5)
```

La variable "self" permet à Python d'accéder aux données de chaque instance de la classe :

```
self.x += 3
```

Self

Pour translater une coordonnée suivant un déplacement (dx, dy), la variable "self" permet d'accéder aux attributs *x* et *y* :

```
def translate(self, dx, dy):
    """
    Deplace le point de coord x et y de dx et de dy
    """
    self.x += dx
    self.y += dy
```

Plan

Surcharge de méthodes

Le langage Python permet de ré-écrire des méthodes spéciales. Par exemple :

- les méthodes pour afficher (print)
- les convertisseurs (int, str, etc.)
- les opérations unaires, arithmétiques (+, *)
- les opérations de comparaison (<, ==, etc.)
- les conteneurs ([], len, in, etc.)

Les méthodes pour afficher 1/2

La fonction **str** est utilisée pour les affichages à destination de l'utilisateur du programme. L'instruction **print(x)** a pour effet d'afficher la chaîne de catactères **str(x)**.

Derrière la fonction **str** se cache une méthode spéciale. L'appel **str(x)** retourne x.__str__(). Cette méthode doit retourner une chaîne de caractères.

Les méthodes pour afficher 2/2

```
def __str__(self):
    txt = "Coord: ["
    txt += str(self.x) + ", "
    txt += str(self.y) + "]"
    return txt
```

```
print (c0)
```

```
Coord: [2.250000, 4.500000]
```

La conversion en valeur booléenne

La conversion en valeur booléenne est également utilisée lorsqu'un objet doit être évalué comme expression booléenne dans une structure if ou while.

```
    def __bool__(self):
        return self.x != 0 and self.y != 0
```

```
c1 = Coord(5, 4)
if c1:
    print("évalué à True")
```

```
??
```

La conversion en valeur booléenne

La conversion en valeur booléenne est également utilisée lorsqu'un objet doit être évalué comme expression booléenne dans une structure if ou while.

```
    def __bool__(self):
        return self.x != 0 and self.y != 0
```

```
c1 = Coord(5, 4)
if c1:
    print("évalué à True")
```

```
évalué à True
```

Surcharge de l'opérateur +

Si les objets doivent pouvoir être utilisés dans des opérations arithmétiques, alors il faut fournir une implémentation pour les méthodes suivantes :

Nom	Méthode	Utilisation
addition	__add__	obj1 + obj2
multiplication	__mul__	obj1 * obj2
soustraction	__sub__	obj1 - obj2
puissance	__pow__(self, o, modulo)	obj1 ** obj2
....	...	...

Ces méthodes prennent toutes en paramètres self et le deuxième opérande de l'opérateur arithmétique

Surcharge de l'opérateur +

Pour le cas de la classe *Coord*, on peut, par exemple, autoriser l'addition de deux coordonnées ou d'une coordonnée avec un scalaire.

```
def __add__(self, c2):
    if isinstance(c2, (int, float)):
        return Coord(self.x + c2, self.y + c2)
    if isinstance(c2, Coord):
        return Coord(self.x + c2.x, self.y +
        ↪  c2.y)
    return NotImplemented
```

Surcharge de l'opérateur $<$

Supposons que l'on veuille avoir une notion d'ordre dans la classe *Coord*. Par exemple on voudrait définir :

$$c1 < c2 \quad \text{si } (c1.x < c2.x) \text{ ou si } (c1.x = c2.x \text{ et } c1.y < c2.y)$$

Comment faire pour surcharger l'opérateur "<" ?

Surcharge de l'opérateur <

On définit la méthode __lt__ dans la définition de la classe :

```
def __lt__(self, other):
    if not isinstance(other, Coord):
        return False
    if self.x < other.x:
        return True
    if self.x == other.x and self.y < other.y:
        return True
    return False
```

On peut alors faire :

```
cA = Coord(1, 0)
cB = Coord(1, 2)
print ('pA < pB : ', pA < pB)
```

cA < cB : True

Surcharge de l'opérateur ==

On dit que deux objets a et b possèdent la même valeur si l'instruction $a == b$ retourne True. Derrière l'opération « == » se cache la méthode `__eq__`.

Le calcul de l'instruction $a == b$ retourne :

```
a.__eq__(b)
```

Quand on dit que a et b ont la même valeur, cela ne veut pas forcément dire que d'un point de vue sémantique a et b ont la même valeur, cela veut dire que la fonction *eq* retourne un objet dont la valeur booléenne est *True*.

Surcharge de l'opérateur ==

Si on ne surcharge pas la méthode *eq*, un objet ne sera égal qu'à lui même.

Quand on écrit une classe, on doit se demander si on veut considérer l'instance comme des valeurs ou des objets. Par exemple, pour comparer des coordonnées, on aurait envie de considérer que deux coordonnées sont égales ssi leurs ordonnées et leurs abscisses sont égales.

Surcharge d'autres opérateurs

Opérateur	Méthode
==	__eq__(self, v)
!=	__ne__(self, v)
<	__lt__
<=	__le__(self, v)
....	...

Les conteneurs

Opérateur	Méthode
len	__len__(self)
o[key]	__getitem__(self, key)
o[key] = value	__setitem__(self, key, value)
key in o	__contains__(self, key)
....	...

Les conteneurs

```
class Coord:
    # .....
    def __len__(self):
        return 2

    def __getitem__(self, k):
        if k == 'x' or k == 0:
            return self.x
        if k == 'y' or k == 1:
            return self.y
        raise KeyError(k)
```

Les conteneurs

```
class Coord:
    # .....

    def __setitem__(self, k, v):
        if not isinstance(v, (int, float)):
            raise TypeError
        if k == 'x' or k == 0:
            self.x = v
        elif k == 'y' or k == 1:
            self.y = v
        else:
            raise KeyError(k)
```

Plan

Variables de classe

Certains attributs ou méthodes, peuvent être partagés entre toutes les classes. Dans ce cas, l'élément n'appartient pas à une instance particulière mais à la classe elle-même. On peut ainsi utiliser ces éléments sans avoir besoin de créer une instance.

Appel

On utilise le point "." depuis le nom d'une classe pour utiliser ou appeler un attribut ou une méthode statique.

Classe Point

```
class Point:
    compteur = 0
    def __init__(self, c):
        self.coord = c
        self.id = Point.compteur
        Point.compteur += 1
```

- La méthode *__init__* de la classe *Point* commence par initialiser l'attribut coord.
- La méthode *__init__* instancie ensuite un identifant (id) en utilisant une variable de classe : ***compteur*** qui appartient à la classe et non pas à l'instance de la classe. Quand on crée une instance de *Point* on ne crée pas une nouvelle instance de ***compteur***. C'est ce qui garantit que les instances de *Point* auront un identifiant unique.

Variables de classe

La propriété *static* transforme une fonction en une méthode statique. La méthode n'a pas besoin de s'exécuter dans le contexte d'une classe. Par conséquent, cette méthode ne prend pas le paramètre *self* comme premier paramètre. Pour déclarer une méthode statique, on utilise le décorateur **staticmethod**.

```
class ObsTime:
  @staticmethod
  def now():
      """Get Current Date and Time
      @return ObsTime """
      nowobj = datetime.now()
      return ObsTime(nowobj.year, nowobj.month,
                     nowobj.day, nowobj.hour,
                     nowobj.minute, nowobj.second)
```

Plan

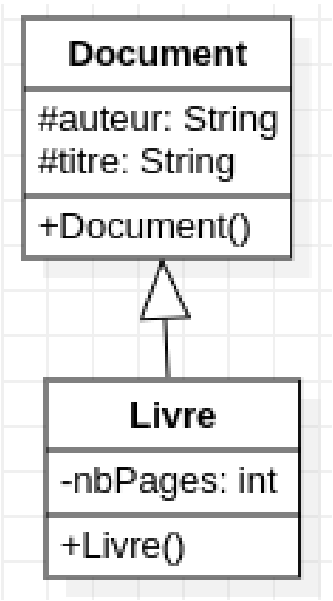
Plan

Héritage

Beaucoup de types structurés ont des propriétés communes avec d'autres types structurés. L'**héritage** fournit un mécanisme qui permet de créer une nouvelle classe à partir d'une classe existante qui correspond à une notion plus abstraite ou plus générale. Cette nouvelle classe possèdera des fonctionnalités différentes et supplémentaires.

Héritage

L'héritage est donc le mécanisme qui permet de traduire une relation de type « est un(e) » et ainsi créer une hiérarchie de types structurés dans laquelle chaque type structuré hérite des attributs des types structurés du dessus.



Héritage

La classe **object** est la classe au sommet de la hiérarchie (rappelez-vous en python, tout est objet).

La classe *Document* hérite de toutes les propriétés de *Object*.

```
class Document:
```

ou

```
class Document(object):
```

Classe Livre

On veut créer un nouveau type *Livre* qui possède les mêmes attributs et certaines méthodes de Document, et en plus d'autres éléments qui lui sont propres.

La classe *Livre* hérite de la classe **parent** *Document*

```
class Livre(Document):
```

Classe Livre

Tout objet de la classe dérivée est donc considéré avant tout comme un objet de la classe mère : un livre est un document. Tout objet de la classe dérivé cumule les attributs dérivés de la classe mère avec ceux définis dans sa propre classe : un livre possède donc un titre et un auteur.

Tous les attributs ou méthodes *x* peuvent être accédés par **self.x** dans le code d'une classe dérivée.

Dans le jargon de la poo, la classe *Livre* est une **sous-classe** de la classe *Document* et hérite donc des attributs de sa **super-classe**

Classe Livre

En plus de l'héritage, on peut :

- ajouter de nouveaux attributs ou méthodes. Par exemple, *Livre* a comme attribut *nbPages* et une méthode *getNbPage*
- surcharger, des attributs ou des méthodes de la superclasse. Par exemple, *Livre* a surchargé les méthodes *__init__* et *__str__*

Héritage et constructeurs

Tout constructeur d'une classe B héritant de A doit débuter par un appel à un constructeur de A.

super().__init__(...) permet d'appeler le constructeur de la classe A(....).

L'appel a *super* doit être la première instruction du constructeur.

Classe Livre

```
class Livre(Document):

    def __init__(self, titre, auteur, nbpage):
        super().__init__(titre, auteur)
        self.nbPages = Point.nbpage
```

- La méthode *__init__* de la classe *Livre* commence par invoquer la méthode *__init__* de la classe *Document* afin d'initialiser les attributs : titre, auteur.

- la méthode *__init__* instancie ensuite l'attribut *nbPages*.

Héritage

Tout objet de la classe dérivée hérite des méthodes de sa classe mère. Seuls les constructeurs ne sont pas hérités.

Classe Livre

- En invoquant l'instruction :

```
lm = Livre("Hugo", "Les misérables", 724)
print str(lm)
```

le système ne voyant pas la méthode *__str__* dans la classe Livre, il va chercher si elle existe dans la classe parente *Document*. Elle existe donc c'est cette méthode qui sera exécutée.

Redéfinition de méthodes

Parfois, le code d'une méthode d'une classe mère peut ne plus être adapté, ne plus faire sens ou manquer de précision pour une de ses classes dérivées. Par exemple, la méthode description de la classe *Document* devrait être adaptée dans la classe *Livre* pour afficher le nombre de pages.

Il est possible de redéfinir dans une classe fille B héritant de A les méthodes héritées de A.

Le type de retour de la méthode redéfinie dans B doit être du même type ou un sous-type du type de retour de la méthode originale définie dans A.

Plan

Encapsulation

L'encapsulation des données est un autre concept de la programmation objet. Il stipule que les données et les fonctions qui opèrent sur elles sont encapsulées dans des objets. Les seules fonctions à être autorisées à modifier les données d'un objet sont les fonctions appartenant à cet objet. Depuis l'extérieur d'un objet, on ne peut le modifier que par des fonctions faisant office d'interface.

Ainsi, il n'est plus à craindre que des fonctions modifient indûment des données. De plus, un changement de structure de données n'entraînera que la réécriture des fonctions associées à l'objet concerné.

Visibilité

En python on réalise la protection des attributs grâce à l'utilisation d'attributs privés. Pour avoir des attributs privés, leur nom doit débuter par __ (deux fois le symbole underscore _).

```
def __init__(self, wkt):
    """ Creation une geometrie """
    self.__listCoord = wktM.createwkt(wkt)
    self.__srid = -1
    self.__centroid = [self.listCoord[0], self.listCoord[1]]
```

Visibilité

Il n'est alors plus possible de faire appel aux attributs *__listCoord*, *__srid* et *__centroid* depuis l'extérieur de la classe Geometrie.

```
pA = Geometrie("POINT ((1 1))")
print (pA.__listCoord)

AttributeError: 'Geometrie' object has no attribute 'listCoo
```

Il faut donc disposer de méthodes qui vont permettre de récupérer ou modifier les informations associées à ces variables.

```
def getListCoord(self):
    return self.listCoord
```

Plan

Polymorphisme

Le troisième principe de base de la poo est le **polymorphisme**. Des fonctions différentes dans des classes différentes peuvent prendre le même nom. Ainsi, dans une hiérachie de classes d'éléments graphiques la fonction dessiner() aura le même nom pour un point ou un polygone. Mais les techniques utilisées pour dessiner ces éléments sont différentes.

Le polymorphisme fait économiser des identificateurs de fonctions et rend les notations plus lisibles. Ainsi, il est plus simple d'écrire :

```
pA.dessiner()
polyDepartement.dessiner()
```

que :

```
pA.dessinerPoint()
polyDepartement.dessinerPolygone().
```

Plan

Plan

Méthode abstraite

Définition

Une méthode est **abstraite** si seul son en-tête est donné (signature), le corps de la méthode est remplacé par *pass*. La méthode abstraite n'a pas d'instruction.

La méthode est annotée *@abstractmethod* avec la librairie *ABC*

Une méthode est déclarée abstraite pour spécifier qu'il s'agit uniquement d'un prototype ou d'une définition. On ne peut pas utiliser directement une classe qui contient des méthodes abstraites : vous devez créer une classe fille qui implémente les méthodes abstraites.

Méthode abstraite

Exemple :

```
from abc import abstractmethod

class ....

    @abstractmethod
    def execute(self, input):
        pass
```

Plan

Classe abstraite

Définition

Une classe *abstraite* est une classe très générale qui décrit des propriétés qui ne seront définies que par des classes héritières, soit parce qu'elle ne sait pas comment le faire, soit parce qu'elle désire proposer différentes mises en œuvres.

Une **classe abstraite** :

- n'est pas instanciable,
- peut avoir des attributs
- peut avoir des méthodes abstraites mais pas nécessairement
- peut avoir des méthodes non abstraites
- déclarée par **metaclass=ABCMeta**.

Classe abstraite

Corollaire

Les classes abstraites permettent de **factoriser** le code : on peut y implémenter des méthodes qui seront communes à plusieurs sous-classes, tout en laissant certaines méthodes *abstract*.

Cela sert à définir les grandes lignes du comportement d'une classe d'objets sans forcer l'implémentation des détails de l'algorithme.

Classe abstraite

On a :

```
class Vehicule:
    // ...

class Voiture(Vehicule):
    // ...

class Velo(Vehicule):
    // ...

class Avion(Vehicule):
    // ...
```

Classe abstraite

On veut ajouter une méthode *deplacement* à toutes ces classes : où la placer ? Difficile de caractériser le déplacement d'un véhicule d'une manière générale ...

De plus, il n'est pas pertinent de pouvoir instancier des objets de type Vehicule : que représentent-ils ? Comment se déplacent-ils ? Quel bruit font-ils ?

Classe abstraite

La solution :

```
class Vehicule(metaclass=ABCMeta):

    # Vehicule n'est plus instanciable !
    @abstractmethod
    def deplacement():
        pass
```

La méthode *deplacement* dans la classe Vehicule est abstraite, c'est une définition. Elle sera implémentée dans les classes filles *Voiture*, *Velo* et *Avion*

Classe abstraite

La solution (suite) :

```
class Voiture(Vehicule):
    def deplacement():
        print("Je roule vite sur 4 roues")

class Velo(Vehicule):
    def deplacement():
        print("Je roule prudemment a deux roues")

class Avion(Vehicule):
    def deplacement():
        print("Je vole")
```

Classe abstraite

On peut rajouter du code dans Vehicule qui sera partagé par les classes qui en héritent.

```
class Vehicule:

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

    @abstractmethod
    def deplacement():
        pass

v = new Voiture();
print(v.vitesseMax)
```

Quand employer des classes abstraites ?

La notion de classe abstraite est intrinsèquement liée à l'héritage, et implique conceptuellement des dépendances fortes de type "est un".

Préférez les classes abstraites lorsque :

- on peut affirmer que "A est un sous-type de B" ;
- on désire factoriser du code, qui sera partagé par des **classes conceptuellement proches** ;