Programmation orientée objet avec Python

Marie-Dominique Van Damme, ENSG

Cycle d'ingénieur - Master mention géomatique
mars 2024

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.

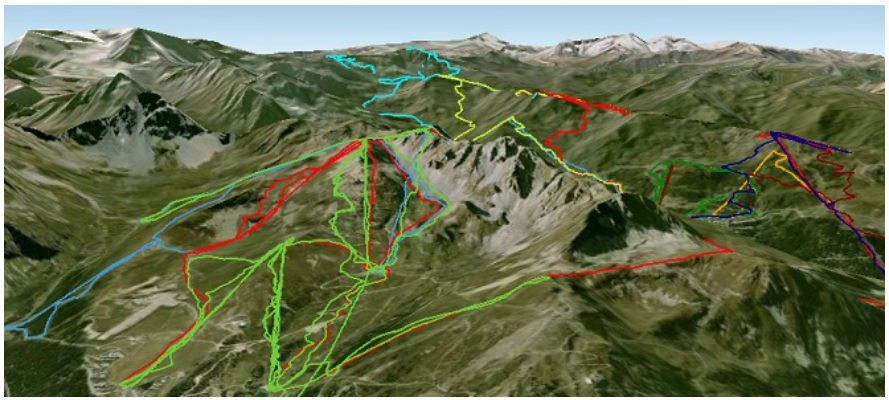
Plan

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 : 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 ?

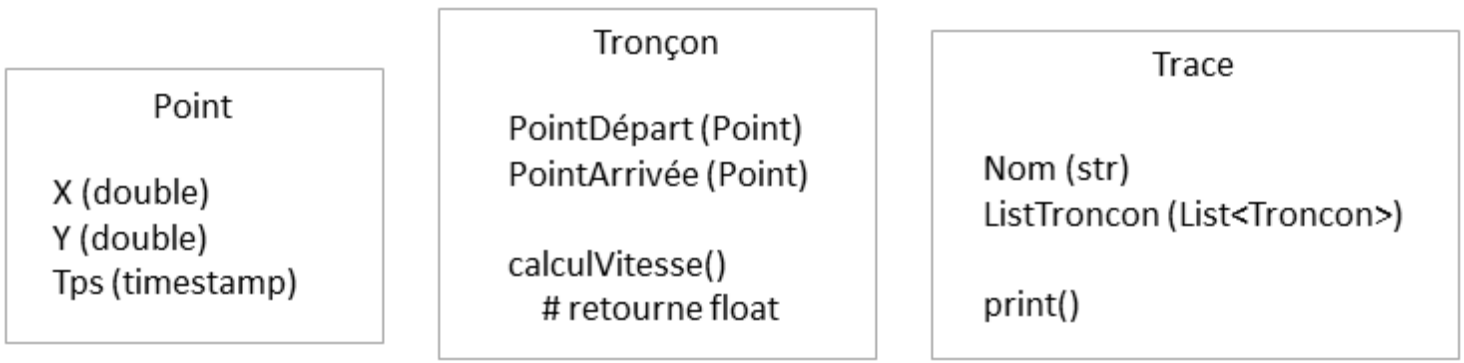


Figure – Types de structures envisagés

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

Imaginons un langage de programmation dans lequel la classe des nombres flottants serait inexistante. On souhaiterait alors implémenter un objet (non-primitif), que l'on appelerait *Float*, et qui ne serait autre qu'un tableau de 2 entiers (l'un faisant référence à la partie entière du nombre, l'autre à sa partie décimale).

On compléterait alors le modèle avec une panoplie de fonctions permettant, d'ajouter, soustraire, multiplier, diviser, comparer ($= \leqslant \geqslant$) deux flottants, de récupérer leur partie entière, leur partie décimale...

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(",")`...

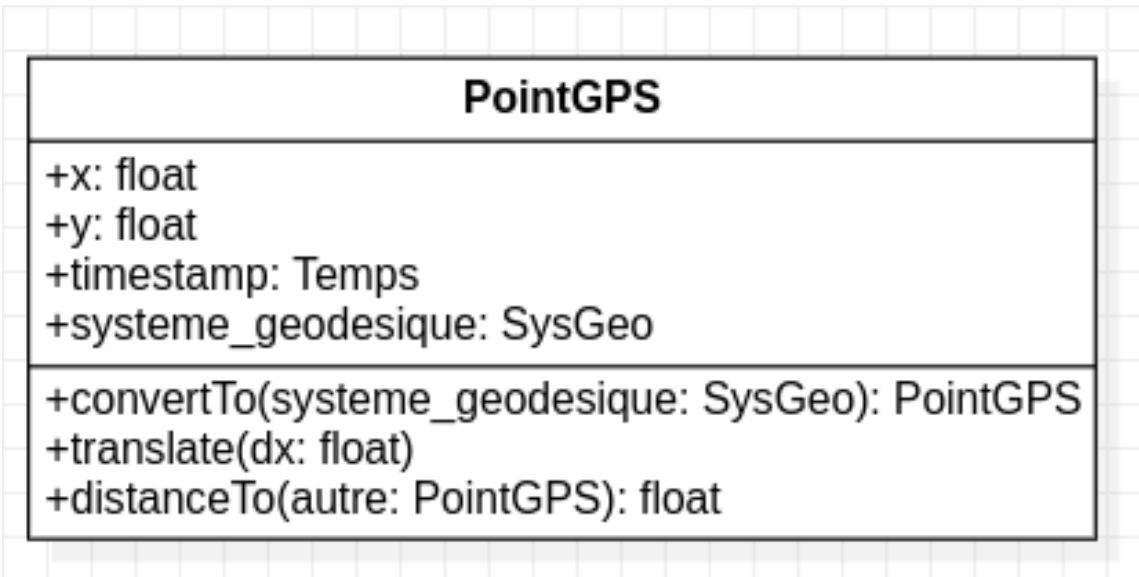
L'existence de deux méthodes de tri sur les listes exprime la dualité entre les deux méthodes de programmation :

`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.

Diagramme de classes

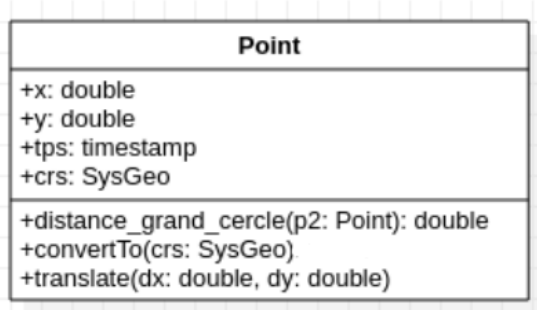
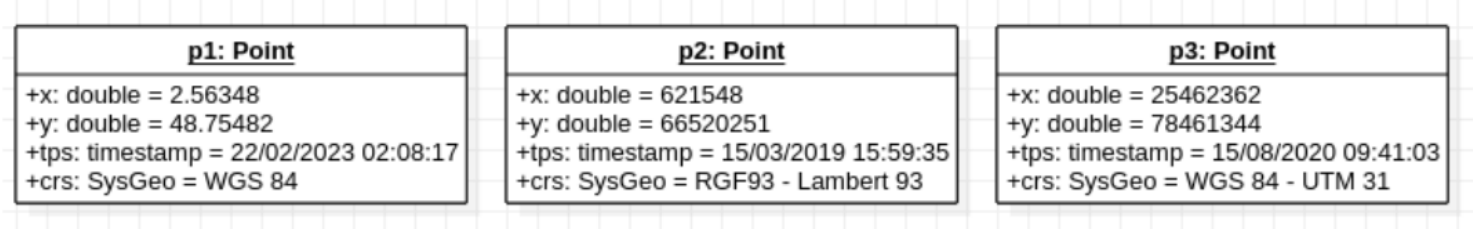
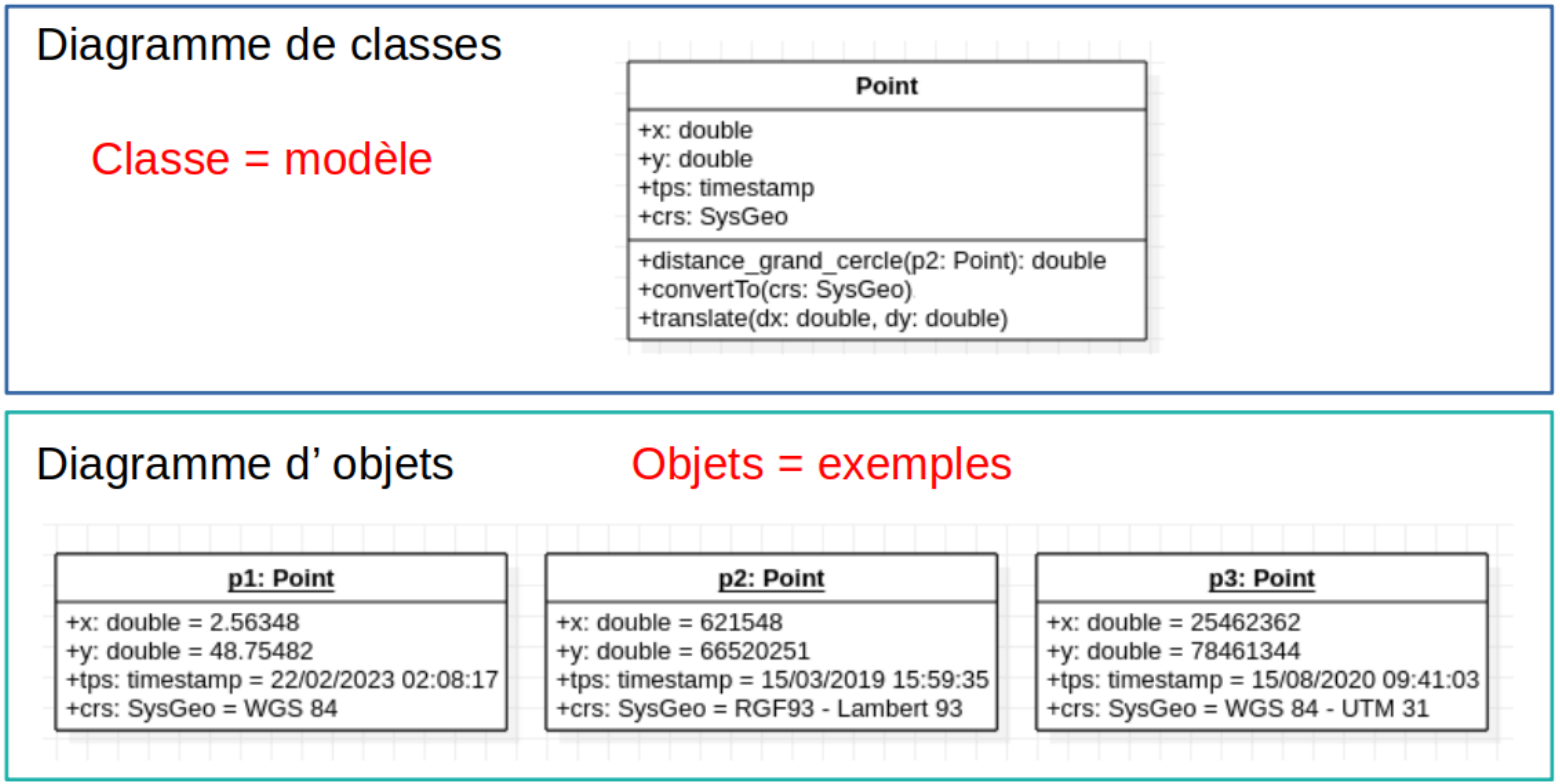


Diagramme d' objets



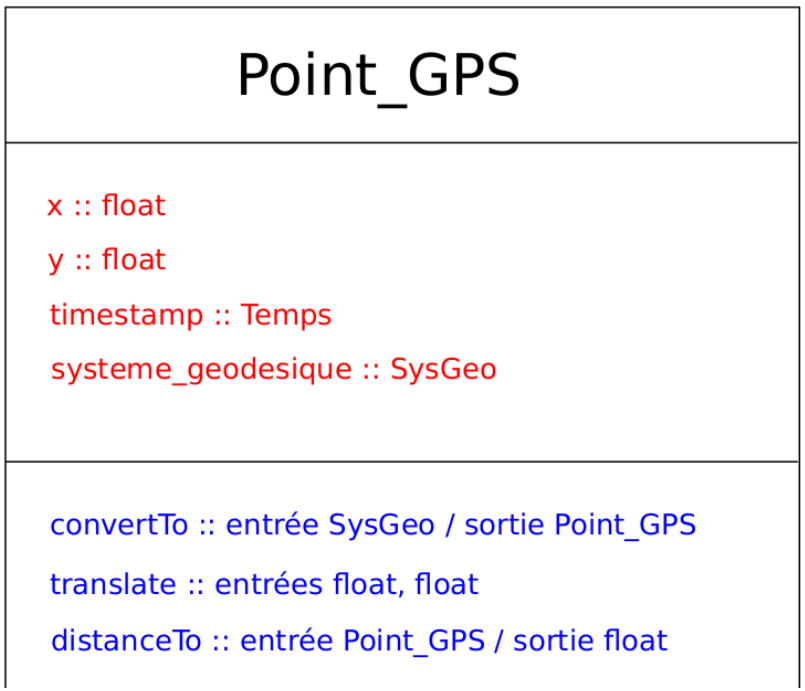
Types de données abstraits



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

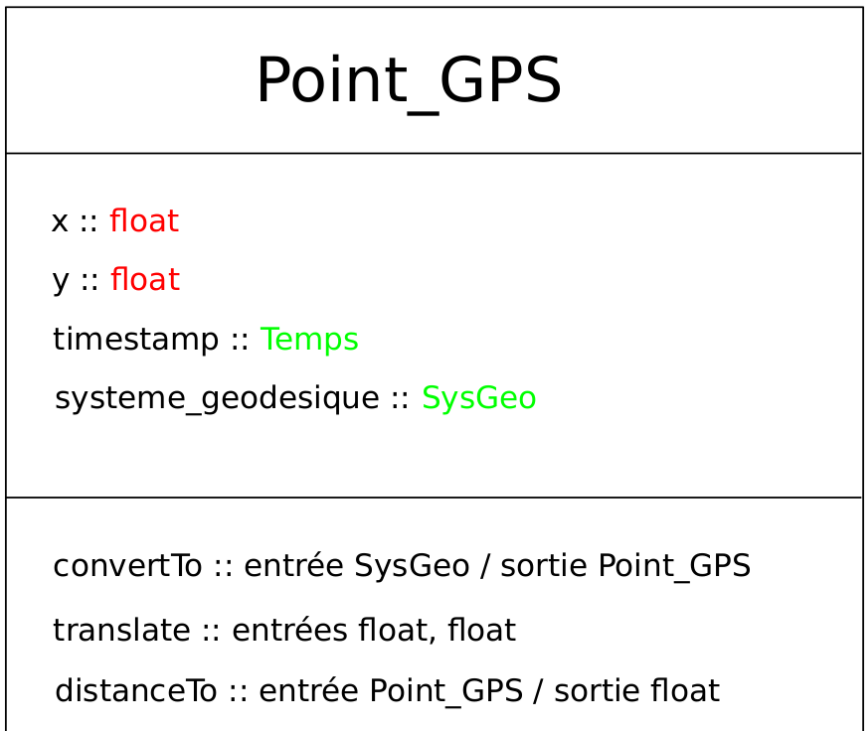
Types de données abstraits

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



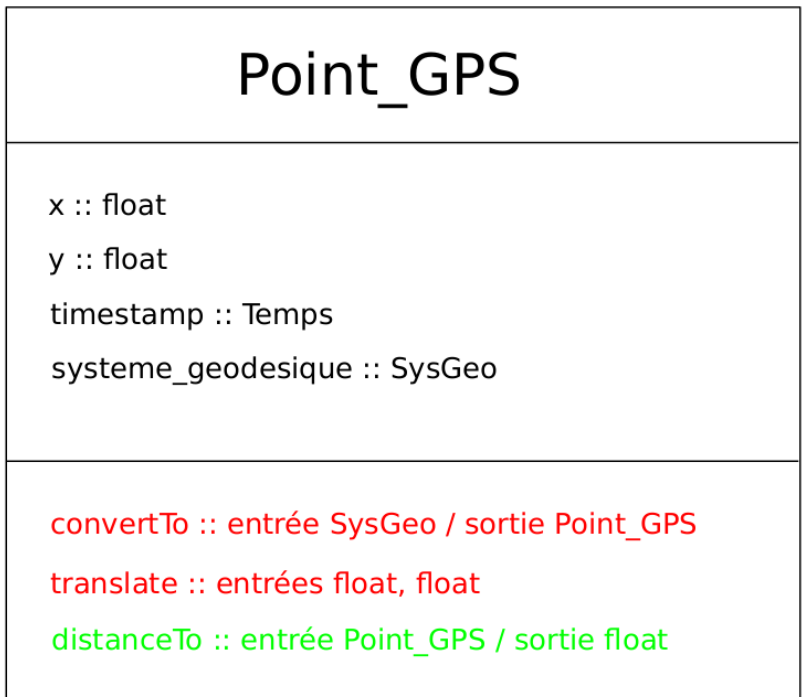
Types de données abstraits

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



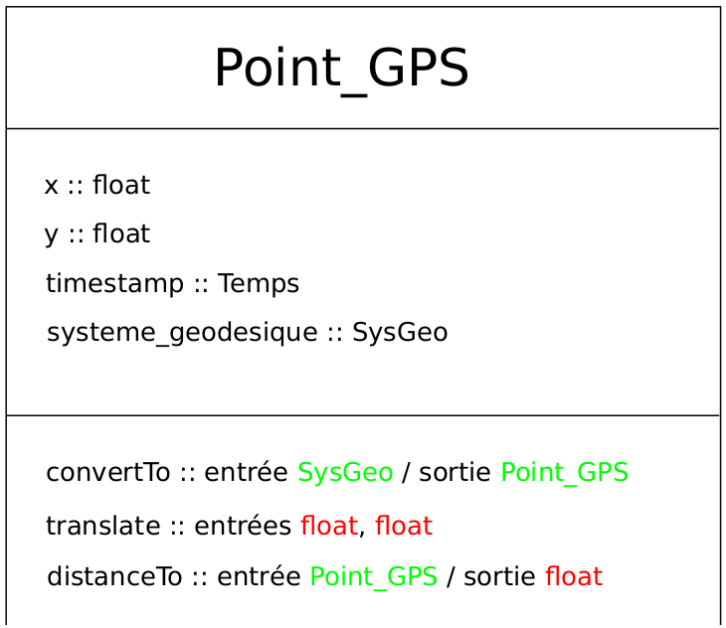
Types de données abstraits

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)



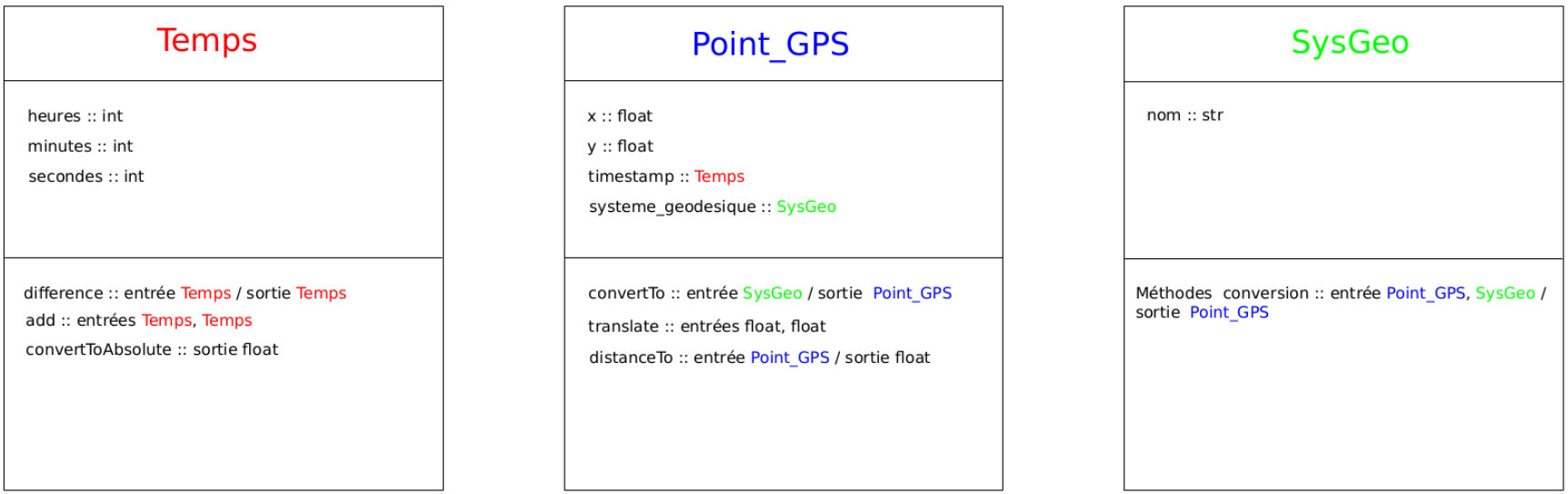
Types de données abstraits

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



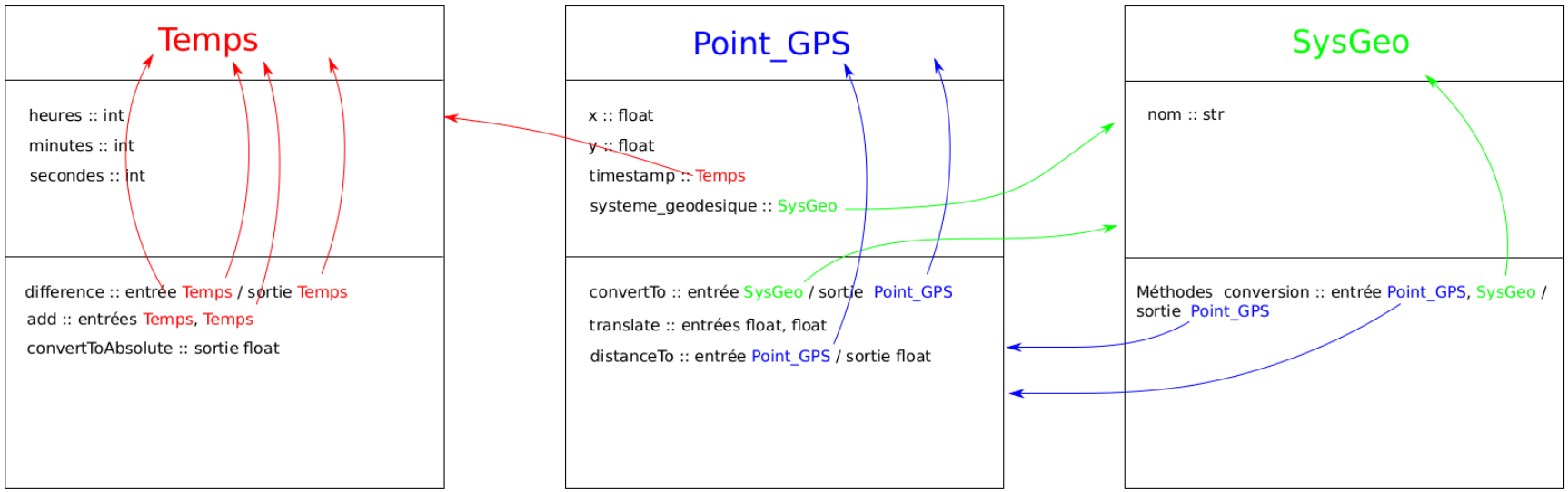
Types de données abstraits

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



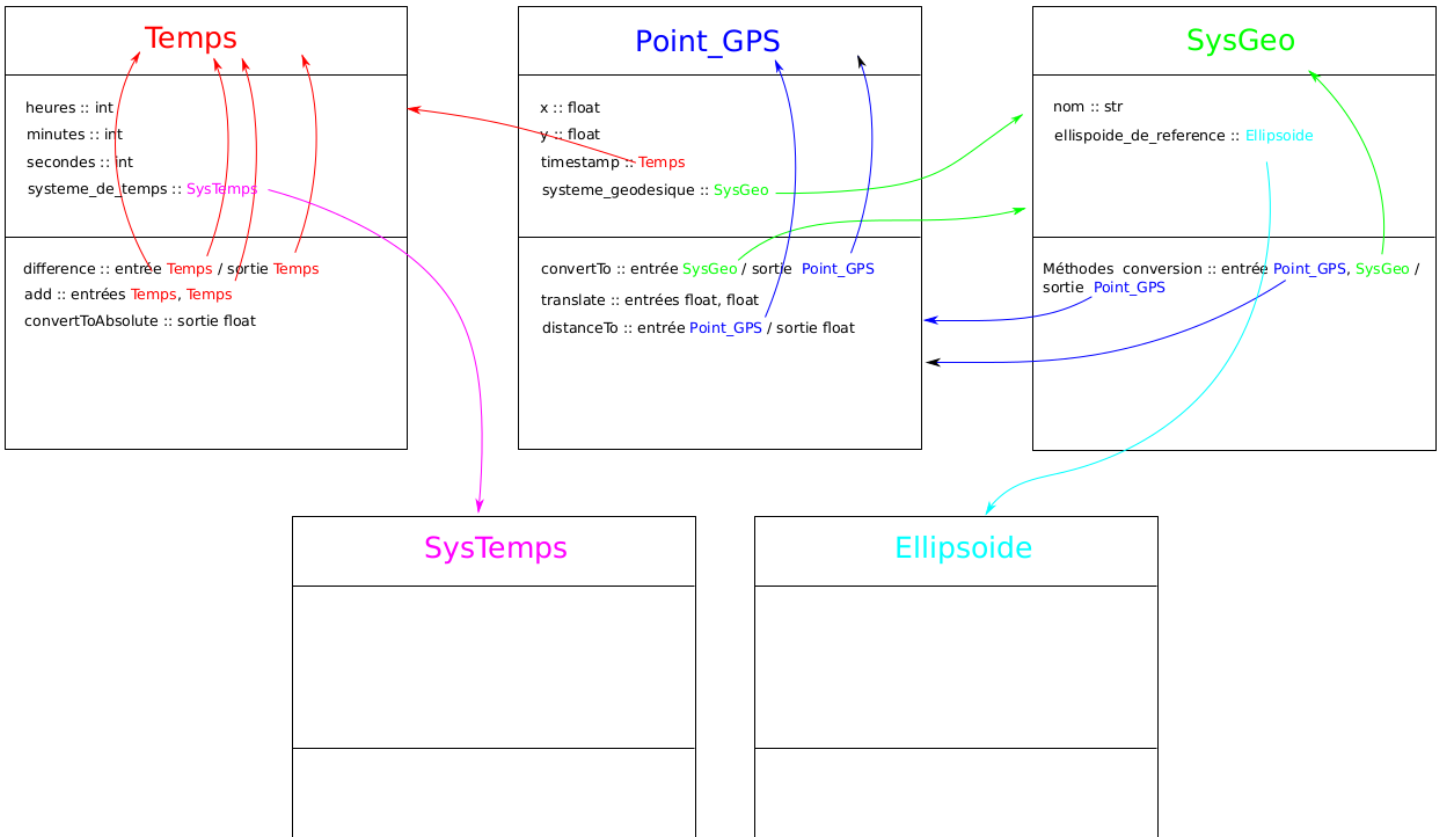
Types de données abstraits

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.

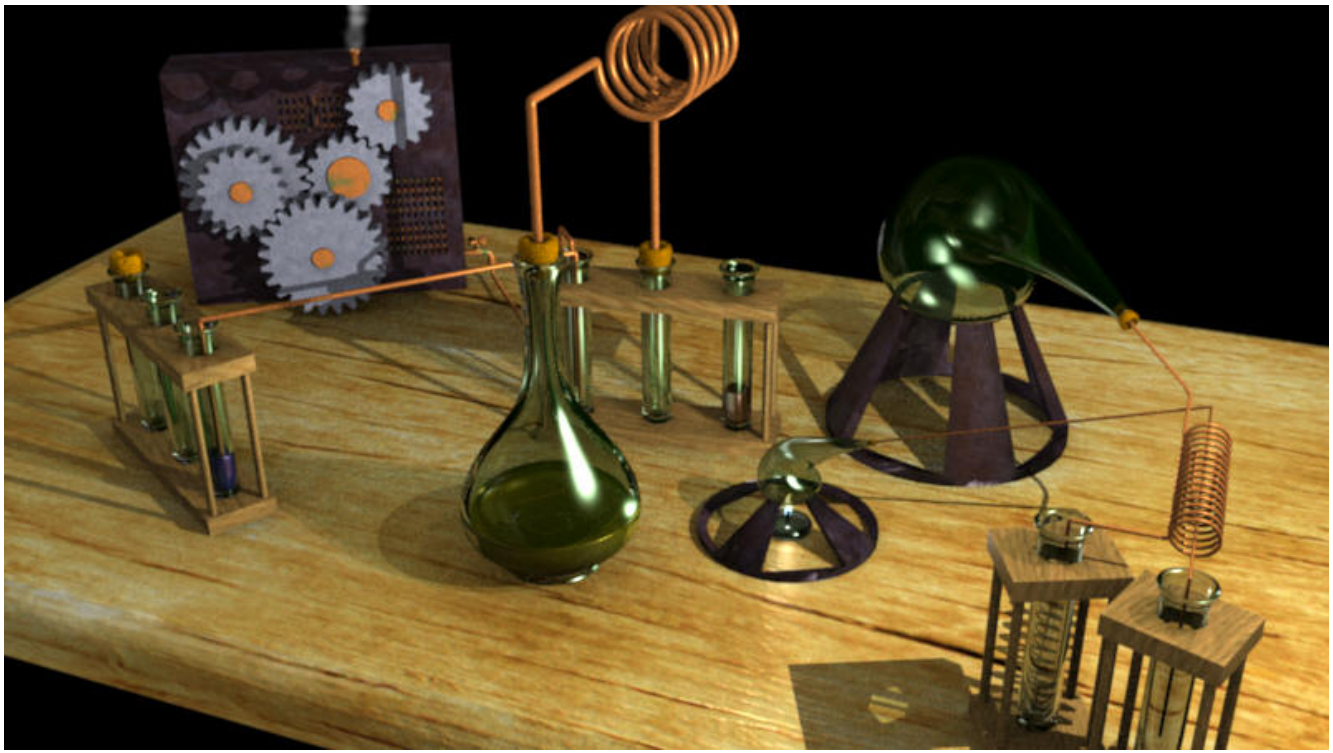
L'interface définit le comportement des opérations : ce qu'elles font, mais sans dire comment elles le font.

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

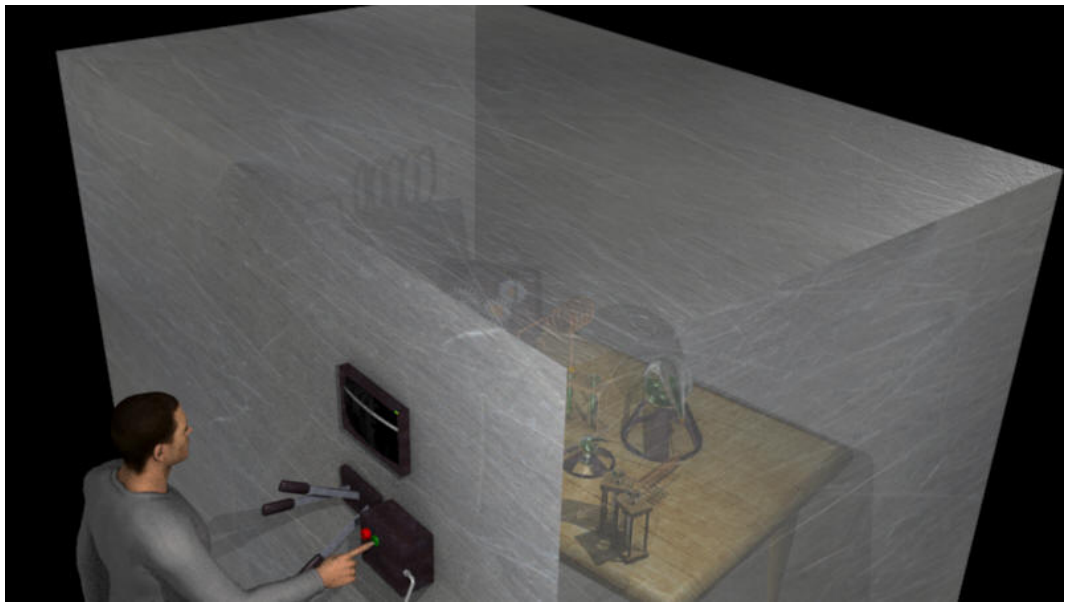
Quand on code un programme, on ajoute beaucoup de variables et de fonctions qui s'entre-appellent :



Référence : Mathieu Nebra, Open classroom

Interfaces

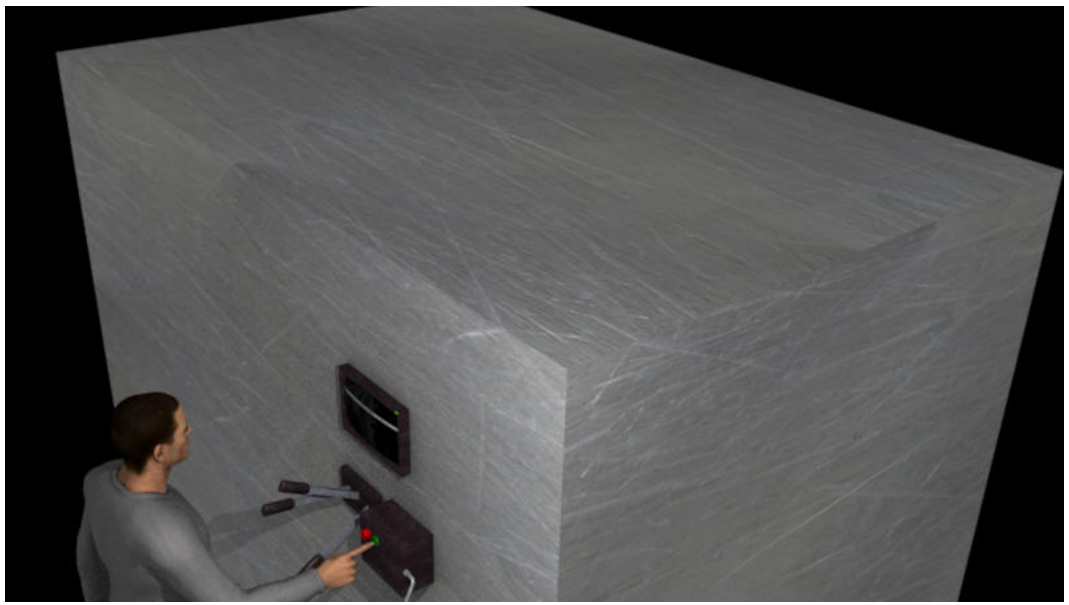
En programmation orientée objet, le code complexe peut-être encapsulé dans des classes qui seront opaques pour l'utilisateur => la complexité est masquée.



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"

Dans l'exemple de la classe Coord, 2 méthodes s'écrivent avec des underscores au début et à la fin du nom de la méthode :

```
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 l'objet :

```
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
```

Premières classes de géométrie

Comme exemple d'utilisation des classes, imaginons qu'il faille concevoir un programme tenant compte des objets géométriques vus auparavant : point, ligne et polygone. On voudrait aussi :

- chaque objet (point, ligne, polygone) ait un constructeur prenant comme paramètre d'entrée une chaine WKT.
- chaque objet est caractérisé par sa projection, et un centroide (centre de la géométrie)

Conception d'un programme

Avant de définir tout un ensemble de structures de données, essayons de voir quelles abstractions pourraient rendre le programme plus efficace.

Existe-t-il des attributs ou des méthodes qui sont en commun pour nos 3 objets : point, ligne et polygone ?

=> ce sont tous des objets géométriques. Ils sont tous définis par une suite finie de points : 1, 2 ou n. Et ils ont des paramètres en commun : la projection et le centroide.

Classe géométrie

Voici une classe "geometrie" qui définit une classe qui contient tout ce qu'on peut avoir en commun :

```
class Geometrie(object):

    def \_\_init\_\_(self, wkt):
        """ Creation une geometrie """
        self.listCoord = createwkt(wkt)
        self.srid = -1
        self.centroid = calculCentroid(self.listCoord)

    def getListCoord(self):
        return self.listCoord

    def \_\_str\_\_(self):
        return str(self.listCoord)
```

Classe géométrie

Quand on veut instancier une classe, il faut regarder la fonction __init__ pour savoir quels paramètres et quelles propriétés sont nécessaires.

Dans notre exemple il s'agit d'une chaine de caractères correspondant au wkt :

```
point = Geometrie("POINT(1 1)")
ligne = Geometrie("LINESTRING(0 2, 2 0)")
poly = Geometrie("POLYGON((20 10, 30 0,
                  40 10, 30 20, 20 10))")
```

Accès aux méthodes

On peut accéder aux informations des instances de classes :

- en utilisant les méthodes associées :

```
point.getListCoord()
```

- ou directement avec l'attribut

```
point.listCoord
```

Les deux instructions vont retourner *[1.0, 1.0]*. La seconde est à éviter pour des raisons que l'on verra plus tard.