LES BASES DE LA PROGRAMMATION ORIENTEE OBJET (EN PYTHON)

PRESENTATION

Dans la Programmation Orientée Objet (**POO**) les programmes sont structurés comme des ensembles d'objets qui interagissent entre eux et avec le monde extérieur.

Le concept de base de la POO est donc la notion d'**objet.**

Comme dans la vie réelle, les objets informatiques peuvent être plus ou moins compliqués, composés de différentes parties qui sont elles-mêmes des objets, faits d'objets plus simples etc.

La **modélisation d'un objet** consiste à :

- Identifier ses **caractéristiques**.
- Identifier les **fonctionnalités** à mettre à disposition pour cet objet.
- Définir les **autorisations d'utilisation des fonctionnalités.**

La **modélisation d'un ensemble d'objets**, nécessite de s'intéresser aux **liens entre les objets.**

PRINCIPE D'ENCAPSULATION

Les fonctionnalités de l'objet et les variables qu'il utilise sont « enfermés » dans l'objet (ils **sont encapsulés**) et sont accessibles à travers des procédures biens définies : l'interface de l'objet.

Une « **classe »** permet de définir un modèle de définition pour un ensemble d'objets de même type.

Les fonctions définies dans une classe sont appelées « **méthodes** ».

Un objet décrit par une classe est appelé une **instance** de cette classe**.** Le mot instance est un anglicisme, qui signifie « cas », « exemple ».

Une classe est décrite par :

- Ses **attributs** – qui sont les caractéristiques des objets de la classe.
- Ses **constructeurs** – pour créer de nouvelles instances d'objets de cette classe.
- Ses **accesseurs** en lecture et en écriture – pour accéder et modifier les attributs de la classe.
- Ses **méthodes** – qui sont les fonctionnalités pour manipuler l'objet de cette classe.
- Les **droits d'usage** des attributs et méthodes.

CLASSES DERIVEES – HERITAGE – POLYMORPHISME

Les classes offrent la possibilité de construire de nouveaux objets à partir d'objets existants, en utilisant sans modification ce qui existe déjà, et en ajoutant des fonctionnalités.

Ce mécanisme de **dérivation** permet de construire une **classe « enfant »** qui **hérite** toutes les propriétés de la **classe « parent »**, et à laquelle on peut :

- par **enrichissement** : ajouter de nouveaux attributs ou nouvelles méthodes,
- par **substitution** : surcharger certaines méthodes.

Le **polymorphisme** permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet en fonction du contexte.

CREATIONS D'UNE CLASSE

Pour créer une nouvelle classe d'objet on utilise l'instruction **class**.

Exemple : définition de la classe « Point() ».

Rappel : en géométrie plane, l'objet « point » est l'ancêtre de tous les objets.

```
class Point(object):
      """Définition d'un point"""
```

Le **double point** et l'**indentation** du bloc d'instruction sont obligatoires.

Le bloc doit contenir au moins une ligne (ici un simple commentaire, une bonne pratique à conserver).

Les parenthèses sont destinées à contenir la référence d'une classe préexistante, pour permettre le mécanisme d'héritage. Lors de la création d'une classe fondamentale (le point est le plus petit objet de la géométrie) la référence est le nom **object**, ancêtre de toutes les classes.

CONSTRUCTEURS

Le constructeur est une méthode spécifique nommée « **`__init__(…)`** » (deux caractères souligné « _ », le mot init, puis encore deux caractères souligné).

Le constructeur a des paramètres pour initialiser les attributs de la classe. Ces paramètres peuvent avoir des valeurs par défaut. Le premier paramètre d'une méthode d'une classe est obligatoirement « **self** ».

```
class Point(object):
      """Classe définissant un point"""
      def __init__(self, x, y) :
```

Les variables x et y sont appelées **attributs d'instance** ou **variables d'instance**, elles ne sont dotées d'aucune méthode.

ACCESSEURS

Pour un objet de base, les principales fonctionnalités sont :

- connaître les informations (**accesseur**)
- modifier les informations (**mutateur**).

En python, la notion de droits d'accès usuellement définie dans les langages de POO (C#, C++, java,..) n'existe pas. Par défaut, les membres d'une classe sont publique.

Il est possible et souhaitable, d'indiquer qu'un membre d'une classe est considéré comme privé en introduisant le symbole souligné « _ ». Ainsi on utilise « **_x** » pour l'attribut « **x considéré comme privé** ».

De plus, il est de préférable de passer par des propriétés (**property**) pour changer les valeurs des attributs. Bien que cela ne soit pas obligatoire en Python, il existe une convention de passer par des **getter** (ou **accesseur** en français) et des **setter** (**mutateurs**) pour changer la valeur d'un attribut.

Les **propriétés** (**property**) permettent de contrôler l'accès à certains attributs d'une instance. Elles se définissent dans le corps de la classe en suivant cette syntaxe :

```
nom_propriete = property(methode_accesseur, methode_mutateur,
  methode_suppression, methode_aide)
```

La **description** d'un objet sous forme de chaîne de caractère se fait avec **`__str__`**

DESTRUCTEUR

En Python le destructeur est implémenté par la méthode **`__del__`**

Exemple : création d'une classe Point()

```
class Point(object):
      """Classe définissant un point,
      caractérisé par ses coordonnées x,y"""
      def __init__(self,x,y):
        self._x = x
        self._y = y

      def _get_x(self):
        """accesseur get pour l'attribut x"""
        return self._x

      def _set_x(self, v):
        """mutateur set pour l'attribut x"""
        self._x  =  v

      def _get_y(self):
        """accesseur get pour l'attribut y"""
        return self._y

      def _set_y(self, v):
        """mutateur set pour l'attribut y"""
        self._y  =  v

      def __str__(self):
        """retourne les coordonnées du point"""
        s="\n(X,Y) = ({},{})".format(self._x,self._y)
        return s;

      def __del__(self):
        """destructeur de la classe, avec message"""
        print("Destruction du point "+str(self)+"\n")

      def affiche(self):
        """permet d'afficher les coordonnees du point"""
        print(str(self))

      coordX = property(_get_x, _set_x)
      coordY = property(_get_y, _set_y)

# Programme principal-----------------------

mon_point_A = Point(0,0)       #création d'un point de coordonnées(0,0)
mon_point_A.affiche()          #affichage des coordonnées
mon_point_A.coordX = 5         #modification de x
mon_point_A.coordY = 10        #modification de y
mon_point_A.affiche()          #affichage des nouvelles coordonnées
del(mon_point_A)               #destruction de l'objet
```

```
>>>
(X,Y) = (0,0)
(X,Y) = (5,10)
Destruction du point (X,Y) = (5,10)
```

DEFINITION D'UNE METHODE

On définit une méthode comme on définit une fonction avec **`def nomfonction(parametres):`** La définition de méthode est placée à l'intérieur de la classe. Le paramètre **`self`** est listé en premier.

```
def affiche(self):
        """permet d'afficher les coordonnees du point"""
        print(str(self))
```

Exemple : création d'une classe cercle()
avec des méthodes pour calculer le diamètre, l'aire et la circonférence.

```
class Cercle(object):
     """Classe définissant un cercle caractérisé par son rayon """
    def __init__(self,rayon):
        self.rayon = rayon

    def diametre(self):
        """Methode permettant de calculer le diamètre du cercle """
        return 2 * self.rayon

    def aire(self):
        """Methode permettant de calculer l'aire du cercle """
        return 3.1416 * self.rayon**2

    def circonference(self):
        """Methode permettant de calculer l'aire du cercle """
        return 3.1416 * 2 * self.rayon
```

HERITAGE

On peut se servir d'une classe existante pour en créer une nouvelle. La nouvelle classe héritera de toutes les propriétés de la classe parent, pourra ajouter ses propres propriétés (surcharge) et modifier certaines propriétés de la classe parent.

Le polymorphisme est un mécanisme qui permet l'application d'une "même" opération (méthode surchargée) à des objets d'une même hiérarchie de classes.

Exemple de classes dérivées sur des objets géométriques

```
class Cercle(object):
    """Classe définissant un cercle caractérisé par son rayon """
    def __init__(self,rayon):
        self.rayon = rayon

    def diametre(self):
        return 2 * self.rayon

    def aire(self):
        return 3.1416 * self.rayon**2

    def circonference(self):
        return 3.1416 * 2 * self.rayon

class Cylindre(Cercle):
    """Classe définissant un cylindre caractérisé par son rayon
    et sa hauteur"""
    def __init__(self, rayon, hauteur):
        Cercle.__init__(self, rayon)
        self.hauteur = hauteur

    def volume(self):
        """la methode aire est heritée de la classe Cercle()"""
        return super().aire() * self.hauteur

    def aire(self):
        """les methode super().aire() et circonference()
        sont heritées de la classe Cercle()"""
        s = super().circonference() * self.hauteur + super().aire() * 2
        return s

class Sphere(Cercle):
    """Classe définissant une sphère caractérisée par son rayon"""
    def __init__(self, rayon):
        Cercle.__init__(self, rayon)

    def aire(self):
        """surcharge de la méthode aire"""
        return super().aire() * 4

    def volume(self):
        """la methode rayon est heritée de la classe Cercle()"""
        return self.aire() * self.rayon/3

class Cone(Cylindre):
    """Classe définissant un cone caractérisé
    par son rayon et sa hauteur"""
    def __init__(self,rayon,hauteur):
        Cylindre.__init__(self,rayon,hauteur)

    def volume(self):
        """surcharge de la méthode volume héritée
        de la classe Cylindre()"""
        return super().volume()/3
```