Fiche POO python

Classes :

- Les classes réunissent des données et des fonctionnalités.
- Créer une nouvelle classe → crée un nouveau *type* d'objet → de nouvelles *instances* de ce type.
- 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).

Une première approche des classes :

Syntaxe de définition des classes :

```
class ClassName :
    <statement-1>
    .
    .
    .
    <statement-N>
```

- Les définitions de classes doivent être exécutées avant d'avoir un effet.
- Vous pouvez placer une définition de classe dans une branche `if` ou à l'intérieur d'une fonction.
- Dans la pratique, les déclarations dans une définition de classe sont généralement des définitions de fonctions mais d'autres déclarations sont permises et parfois utiles.

Objets classes :

- Les objets classes ont deux types d'opérations :
  - des références à des attributs.
    - Ils utilisent la syntaxe standard utilisée pour toutes les références d'attributs en Python : `obj.nom`.

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

    - MyClass.i et MyClass.f sont des références à des attributs valident.
    - On peut changer la valeur des attributs par des affectations.
  - l'instanciation.

- L'*instanciation* de classes utilise la notation des fonctions.
- Considérez simplement que l'objet classe est une fonction sans paramètre qui renvoie une nouvelle instance de la classe.

```
x = MyClass()
```

- Crée une nouvelle *instance* de la classe et affecte cet objet à la variable locale `x`.
- Les classes peuvent définir une méthode spéciale appelés `__ini__()`.

```
def __init__(self) :
    self.data = []
```

- Une classe avec la méthode `__init__()` , fait appelle automatiquement à cette méthode à la création d'une nouvelle instance.
- `__init__()` peut contenir plusieurs arguments.

```
>>> class Complex:
        def __init__(self, realpart, imagpart):
            self.r = realpart
            self.i = imagpart

>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Objets instances :

- Les seules opérations comprises par les objets instances sont des références d'attributs.
- Il y a deux sortes de noms d'attributs valides :
  - les attributs 'données'.

        - Corresponds à des variables d’instances.
        - N’ont pas besoin d’être déclarés.
    - les méthodes.
        - Une méthode est une fonction qui "appartient à" un objet.
        - Tous les attributs d'une classe qui sont des objets fonctions définissent les méthodes correspondantes de ses instances.
        - Dans notre exemple, `x.f` est une référence valide à une méthode car `MyClass.f` est une fonction, mais pas `x.i` car `MyClass.i` n'en est pas une. Attention cependant, `x.f` n'est pas la même chose que `MyClass.f` --- Il s'agit d'un *objet méthode*, pas d'un objet fonction.

Objets méthode :

- Une méthode est appelée juste après avoir été liée.

```
class Point:
    def deplace(self, dx, dy): # Objets méthode
        self.x = self.x + dx
        self.y = self.y + dy

a = Point()
a.x = 1
a.y = 2
print("a : x =", a.x, "y =", a.y)
a.deplace(3, 5)
print("a : x =", a.x, "y =", a.y)
```

Classes et variables d’instance :

- Les variables d'instance stockent des informations relatives à chaque instance.
- Les variables de classe servent à stocker les attributs et méthodes communes à toutes les instances de la classe.

```
class Dog:

    kind = 'canine'         # class variable shared by all insta

    def __init__(self, name):
        self.name = name    # instance variable unique to each i

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

- les données partagées mutable (telles que les listes, dictionnaires, etc.) peuvent avoir des effets surprenants.

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
```

```
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

- Une conception correcte de la classe est d'utiliser une variable d'instance à la place.

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

Héritage :

- En POO, “hériter” signifie “avoir également accès à”. Un objet “hérite” des méthodes de la classe qui l’a défini → l’objet peut utiliser ces méthodes.
- La notion d’héritage est intéressante lorsqu’on va l’implémenter entre deux classes. En Python, nous allons pouvoir créer des “sous-classes” ou des classes “enfants” à partir de classes de base ou classes “parentes”.
- La syntaxe pour définir une sous-classe à partir d’une classe de base est la suivante :

```python
class Utilisateur :
    anciennete = 0

    def __init__(self, nom, age) :
        self.user_name = nom
        self.user_age = age

    def getName(self) :
        print("Salut, je suis", self.getName)

class Client(Utilisateur) :
    is_client = True

jerome = Client("Jerome", 20)
print(jerome.user_age)
print(jerome.is_client)

```

- Ici, `Utilisateur` est notre classe de base et `Client` est notre sous-class.
- La sous-classe hérite des variables et fonctions de la classe parent (et notamment de sa fonction `__init__()`) et peut également définir ses propres variables et fonctions.
- On va pouvoir instancier la classe enfant pour créer de nouveaux objets et ces objets vont avoir accès aux variables et fonctions définies dans la sous-classe et dans la classe de base.

```
JeromeYU@Jeromes-Air LU2IN013 Projet % python -u test.py
20
True
```

La surcharge des méthodes de classe :

- “Surcharger” une méthode signifie la redéfinir d’une façon différente.
- Les classes filles ou sous-classes vont pouvoir surcharger les méthodes héritées de leur classe parent et définir des variables de même nom que celles de leur classe parent.
- On ne surcharge que les méthodes car pour les variables, définir une variable avec le même nom dans la sous-classe correspond à créer une variable locale à la sous-classe en plus de celle “globale” (celle disponible dans la classe de base) mais ces deux variables sont différentes tandis que lorsqu’on réécrit une méthode la nouvelle méthode remplace véritablement l’ancienne pour les objets de la sous-classe.
- Réécrivons notre classe `Client()` afin de définir certaines variables locales et de surcharger les méthodes de la classe mère `Utilisateur` :

```python
class Client(Utilisateur) :
    is_client = True
    anciennete = 10

    def __init__(self, nom, age, mail) :
        self.user_name = nom
        self.user_age = age
        self.user_mail = mail

    def getName(self):
        print("Je suis", self.user_name, ". Mon mail :", self.user_mail)

jerome = Client("Jerome", 20, "yujerome44@gmail.com")
jerome.getName()
print(jerome.anciennete)
print(jerome.is_client)
```

```
JeromeYU@Jeromes-Air LU2IN013 Projet % python -u test.py
Je suis Jerome . Mon mail : yujerome44@gmail.com
10
True
```

- On ne voudra souvent pas simplement réécrire l’intégralité du code d’une classe mère dans une classe fille (cela signifierait que nos classes sont mal construites) mais on voudra “étendre” la méthode, lui rajouter des instructions spécifiques pour une classe fille.

- On va pouvoir faire cela en appelant la méthode de notre classe mère depuis notre classe fille avec la syntaxe `ClasseDeBase.nomDeMethode()`, ce qui va nous permettre de récupérer l'intégralité du code de la classe mère.
- Modifions à nouveau notre classe `Client` afin d'étendre la fonction `__init__()` de la classe mère :

```python
class Utilisateur :
    anciennete = 0

    def __init__(self, nom, age) :
        self.user_name = nom
        self.user_age = age

    def getName(self) :
        print("Salut, je suis", self.user_name)

class Client(Utilisateur) :
    is_client = True
    anciennete = 10

    def __init__(self, nom, age, mail) :
        Utilisateur.__init__(self, nom, age)
        self.user_mail = mail

    def getName(self):
        print("Je suis", self.user_name, ". Mon mail :", self.user_mail)

jerome = Client("Jerome", 20, "yujerome44@gmail.com")
jerome.getName()
print(jerome.anciennete)
print(jerome.is_client)

```

```
JeromeYU@Jeromes-Air LU2IN013 Projet % python -u test.py
Je suis Jerome . Mon mail : yujerome44@gmail.com
10
True
```

Polymorphisme :

- "Polymorphisme" signifie littéralement "plusieurs formes".
- Capacité d'une variable, d'une fonction ou d'un objet à prendre plusieurs formes, posséder plusieurs définitions différentes.
- Pour bien comprendre ce concept, imaginons qu'on définisse une classe nommée `Animaux` qui possède des fonctions comme `seNourrir()`, `seDeplacer()`, etc. Notre classe va pouvoir ressembler à ça :

```python
class Animaux :
    def __init__ (self, nom) :
        self.animal_nom = nom

    def seNourrir(self):
        faim = True
        while faim == True:
            self.manger = True #Etc etc

    def seDeplacer (self) :
        pass
```

- Ma classe `Animaux` dispose donc d'une fonction `seDeplacer()` qui ne contient pas d'instruction. Maintenant, nous allons créer des sous-classes de `Animaux` pour différents animaux : `Chien`, `Aigle` et `Dauphin` par exemple.
- Ces trois sous classes vont par défaut hériter des membres de leur classe mère `Animaux` et notamment de la méthode `seDeplacer()`. Chacune des sous classes va implémenter cette méthode différemment, c'est-à-dire va la définir différemment.
- Pour ma classe `Chien` par exemple, la méthode `seDeplacer()` va renvoyer une valeur "courir" tandis que pour `Aigle` cette méthode va renvoyer une valeur "voler". Pour `Dauphin`, `seDeplacer()` renverra "nager".

Héritage multiple :

- Python gère également une forme d'héritage multiple. Une définition de classe ayant plusieurs classes mères est de cette forme :
- La classe fille va chercher les attributs hérités des classes mère en recherche en profondeur, de la gauche vers la droite

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
```

```
    .
    <statement-N>
```

Variables privées :

- On réalise la protection des attributs de notre classe `Point` grâce à l'utilisation d'attributs privées. Pour avoir des attributs privés, leur nom doit débuter par `__`.

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

- il n'est alors plus possible de faire appel aux attributs `__x` et `__y` depuis l'extérieur de la classe `Point`.
- Il faut donc disposer de méthodes qui vont permettre par exemple de modifier ou d'afficher les informations associées à ces variables.

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def deplace(self, dx, dy):
        self.__x = self.__x + dx
        self.__y = self.__y + dy

    def affiche(self):
        print("abscisse =", self.__x, "ordonnee =", self.__y)

a = Point(2, 4)
a.affiche()
```

```
a.deplace(1, 3)
a.affiche()
```

Accesseurs et mutateurs :

- Les **accesseurs** qui fournissent des informations les valeurs de certains de ses attributs (généralement privés) sans les modifier.
- • les **mutateurs** qui modifient les valeurs de certains de ses attributs.
- On rencontre souvent l'utilisation de noms de la forme `get_XXXX()` pour les accesseurs et `set_XXXX()` pour les mutateurs.

```
class Point:
    def __init__(self, x, y):
        self.set_x(x)
        self.set_y(y)

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x

    def get_y(self):
        return self.__y

    def set_y(self, y):
        self.__y = y

a = Point(3, 7)
print("a : abscisse =", a.get_x())
print("a : ordonnee =", a.get_y())
a.set_x(6)
a.set_y(10)
```

```
print("a : abscisse =", a.get_x())
print("a : ordonnee =", a.get_y())
```

- L'utilisation d'un mutateur autorise la possibilité d'effectuer un contrôle sur les valeurs de l'attribut. (Par exemple, il serait possible de n'autoriser que des valeurs positives pour les attributs privés __x et __y)
- Notez qu'il n'est pas toujours prudent de prévoir une méthode d'accès pour chacun des attributs privés d'un objet. En effet, il ne faut pas oublier qu'il doit toujours être possible de modifier l'implémentation d'une classe de manière transparente pour son utilisateur.
- Il existe en Python une autre approche pour gérer ce type de situation. Elle utilise le décorateur `@property`.

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

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, x):
        self._x = x

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, y):
        self._y = y

a = Point(3, 7)
```

```
print("a : abscisse =", a.x)
print("a : ordonnee =", a.y)
a.x = 6
a.y = 10
print("a : abscisse =", a.x)
print("a : ordonnee =", a.y)
```

Trucs et astuces :

- Il est parfois utile d'avoir un type de donnée similaire au *record* du Pascal ou au *struct* du C, qui regroupent ensemble quelques attributs « données » nommés. L'approche idiomatique correspondante en Python est d'utiliser des `dataclasses` :

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int

>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

Attributs de classe :

```
class A:
    nb = 0 # Attribut de classe
```

```
    def __init__(self, x):
        print("creation objet de type A")
        self.x = x
        A.nb = A.nb + 1

print("A : nb = ", A.nb)
print("Partie 1")
a = A(3)
print("A : nb = ", A.nb)
print("a : x = ", a.x, " nb = ", a.nb)
print("Partie 2")
b = A(6)
print("A : nb = ", A.nb)
print("a : x = ", a.x, " nb = ", a.nb)
print("b : x = ", b.x, " nb = ", b.nb)
c = A(8)
print("Partie 3")
print("A : nb = ", A.nb)
print("a : x = ", a.x, " nb = ", a.nb)
print("b : x = ", b.x, " nb = ", b.nb)
print("c : x = ", c.x, " nb = ", c.nb)
```

Méthodes de classe :

```
class A:
    nb = 0

    def __init__(self):
        print("creation objet de type A")
        A.nb = A.nb + 1
        print("il y en a maintenant ", A.nb)

    @classmethod
```

```
    def get_nb(cls): # Méthode de class
        return A.nb

print("Partie 1 : nb objets = ", A.get_nb())
a = A()
print("Partie 2 : nb objets = ", A.get_nb())
b = A()
print("Partie 3 : nb objets = ", A.get_nb())
```

- Pour créer une méthode de classe, il faut la faire précéder d'un : `@classmethod`
- Le premier argument de la méthode de classe doit être `cls`.