

Les classes sous Python

Définition, implémentation, instanciación

Python intègre des particularités – pour ne pas dire bizarreries – que l'on ne retrouve pas dans les langages objets populaires (C++, Java, C#). Ex. création à la volée d'une propriété sur un objet (instance de classe), possibilité de l'utiliser dans les méthodes (alors qu'elle n'apparaît nulle part dans la définition de la classe).

Par conséquent, pour forger les bonnes habitudes, ce support adopte la présentation la plus conventionnelle possible par rapport aux autres langages précités, quitte à passer sous silence certaines possibilités de Python.

Elaboration et instanciation

LES CLASSES SOUS PYTHON

Une **classe** est un type permettant de regrouper dans la même **structure** : les informations (**champs**, propriétés, attributs) relatives à une entité ; les procédures et fonctions permettant de les manipuler (**méthodes**). Champs et méthodes constituent les **membres** de la classe.

Remarques :

- La classe est un type structuré qui va plus loin que l'enregistrement (ce dernier n'intègre que les champs)
- Les champs d'une classe peuvent être de type quelconque
- Ils peuvent faire référence à d'instances d'autres classes

Termes techniques :

- « Classe » est la structure ;
- « Objet » est une instance de la classe (variable obtenue après **instanciation**) ;
- « Instanciation » correspond à la création d'un objet
- L'objet est une référence (traité par le ramasse-miettes, destruction explicite inutile)

Ce n'est pas obligatoire, mais on a toujours intérêt à définir les classes dans des modules. On peut avoir plusieurs classes dans un module.

Ex. **ModulePersonne.py**

```
#début définition
class Personne:
    """Classe Personne"""

    #constructeur
    def __init__(self):
        #lister les champs
        self.nom = ""
        self.age = 0
        self.salaire = 0.0
    #fin constructeur

#fin définition
```

- **class** est un mot clé permettant de définir la structure
- **Personne** est le nom de la classe ici
- **"""Classe Personne"""** sert à documenter la classes
- Regarder le rôle des « : » et des **indentations**
- **__init__** est une méthode standard appelée **constructeur**, automatiquement appelée lors de l'instanciation
- **self** représente l'instance elle-même, **elle doit apparaître en première position dans la définition de toutes les méthodes**, mais il ne sera pas nécessaire de la spécifier lors de l'appel
- nous exploitons le constructeur pour énumérer les champs de la classe (pas obligatoire, mais bonne habitude à prendre), ils sont non typés
- le constructeur peut prendre des paramètres en entrée. Ex. initialisation des champs
- contrairement aux autres langages, un seul constructeur par classe seulement
- noter le rôle du « . » dans l'accès aux champs

```
# -*- coding: utf -*-
```

```
#appel du module
```

```
import ModulePersonne as MP
```

```
#instanciation
```

```
p = MP.Personne()
```

```
#affiche tous les membres de p
```

```
print(dir(p))
```

```
#affectation aux champs
```

```
p.nom = input("Nom : ")
```

```
p.age = int(input("Age : "))
```

```
p.salaire = float(input("Salaire : "))
```

```
#affichage
```

```
print(p.nom, ", ", p.age, ", ", p.salaire)
```

- Il faut d'abord **importer** le module contenant la classe, nous lui attribuons l'alias **MP** ici
- Pour la création de l'instance **p**, nous spécifions le module puis le nom de la classe
- Il y a **()** parce que c'est bien une méthode que nous appelons à la création : le constructeur
- Le paramètre **self** du constructeur n'est pas à spécifier sur l'instance
- Noter le rôle de « **.** » lors de l'accès aux champs de l'instance
- Les champs sont accessibles en lecture / écriture
- L'accès direct aux champs n'est pas conseillé en programmation objet, **mais on fera avec cette année** (sous Python, pour rendre un attribut privé, il faut mettre un double **_** devant le nom du champ et créer alors des accesseurs et des mutateurs)

#début définition

class Personne:

"""Classe Personne"""

#constructeur

def **__init__**(self):

#lister les champs

self.nom = ""

self.age = 0

self.salaire = 0.0

#fin constructeur

#saisie des infos

def **saisie**(self):

self.nom = input("Nom : ")

self.age = int(input("Age : "))

self.salaire = float(input("Salaire : "))

#fin saisie

#affichage des infos

def **affichage**(self):

print("Son nom est ", self.nom)

print("Son âge : ", self.age)

print("Son salaire : ", self.salaire)

#fin affichage

#fin définition

Nous avons implémenté 2 méthodes supplémentaires dans la classe.

Le programme principal s'en trouve grandement simplifié.



-*- coding: utf -*-

#appel du module

import ModulePersonne **as** MP

#instanciation

p = MP.Personne()

#saisie

p.saisie()

#méthode affichage

p.affichage()

Remarque : Noter le comportement de **self**

Rajouter la méthode `retraite()` qui calcule le nombre d'années avant l'âge `limite` de la retraite.

```
#début définition
class Personne:
    """Classe Personne"""

    ...

#reste avant retraite
def retraite(self, limite):
    reste = limite - self.age
    if (reste < 0):
        print("Vous êtes à la retraite")
    else:
        print("Il vous reste %s années" % (reste))
#fin retraite

#fin définition
```



Programme principal

```
# -*- coding: utf -*-
#appel du module
import ModulePersonne as MP
#instanciation
p = MP.Personne()
#saisie
p.saisie()
#méthode affichage
p.affichage()
#reste avant retraite
p.retraite(62)
```



Un exemple

```
>>>
Nom : Toto
Age : 51
Salaire : 1200
Son nom est  Toto
Son âge :  51
Son salaire :  1200.0
Il vous reste 11 années
>>> |
```

Saisie

Affichage

Retraite

Gérer une collection d'objets – L'exemple des listes

COLLECTION D'OBJETS

```
# -*- coding: utf -*-

#appel du module
import ModulePersonne as MP

#liste vide
liste = []

#nb. de pers ?
n = int(input("Nb de pers : "))

#saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)

#affichage
print("*** début affichage 1")
for p in liste:
    print("-----")
    p.affichage()
```

Créer l'objet référencé par **a**, effectuer la saisie. Ajouter la référence dans la **liste**.

Le typage est automatique, **p** est bien de type **Personne**.

Python propose des outils pour la gestion des collections d'objets hétérogènes (tuple, liste, dictionnaire). Ils sont opérationnels pour les instances de nos classes.

Exemple d'exécution

```
Nb de pers : 2
Nom : Toto
Age : 35
Salaire : 1200
Nom : Tata
Age : 36
Salaire : 3000
*** début affichage 1
-----
Son nom est  Toto
Son âge :   35
Son salaire : 1200.0
-----
Son nom est  Tata
Son âge :   36
Son salaire : 3000.0
```

Vérifier toujours que l'indice utilisé est valide.

```
#accès par numéro
numero = int(input("N° ind. à traiter :"))
if (numero < len(liste)):
    b = liste[numero]
    b.salaire = b.salaire * 2
    #affichage de nouveau
    print("xxx début affichage 2")
    for p in liste:
        print("-----")
        p.affichage()
else:
    print("indice non valable")
```

Récupération de la référence de l'objet, indice = numéro

Accès au champ **salaire** de l'objet référencé par **b**.

liste[numero].salaire = ...
fonctionne également !

Exemple d'exécution

```
N° ind. à traiter :1
xxx début affichage 2
-----
Son nom est  Toto
Son âge :   35
Son salaire : 1200.0
-----
Son nom est  Tata
Son âge :   36
Son salaire : 6000.0
```

- Les collections de nos objets (instances de nos classes) ouvre la porte à une programmation organisée et efficace
- Leur intérêt décuplera lorsque nous étudierons les fichiers
- Il est judicieux d'élaborer une classe dédiée à la gestion de la collection (où la liste serait un champ de la classe)
- Les objets de la collection peuvent être différents. Quand ce sont des instances de classes **héritières** du même ancêtre, on parle de liste polymorphe
- Le gestionnaire de collection peut être un dictionnaire, le mécanisme « clé – valeur » (« clé – objet » en l'occurrence pour nous) ouvre des possibilités immenses (ce mécanisme est très en vogue, ex. bases NoSQL)

Héritage et surcharge des méthodes, variables de classes

PLUS LOIN AVEC LES CLASSES

Une fonction peut renvoyer un objet (instance de classe)

```
#début définition
class Personne:
    """Classe Personne"""
    ...
    #copie des infos
    def copie(self):
        q = Personne()
        q.nom = self.nom
        q.age = self.age
        q.salaire = self.salaire
        return q
    #fin copie

#fin définition
```

Une fonction peut renvoyer
une instance de classe

← **q** est une nouvelle instance de la classe **Personne**

← La fonction renvoie l'objet **q**

```
#instanciation
p = MP.Personne()
#saisie
p.saisie()
print(">> Affichage de p")
p.affichage()
#copie
q = p.copie()
print(">> Affichage de q")
q.affichage()
#comp. références
pareil = (p == q)
print("ref. identiques : %s" % (pareil))
```

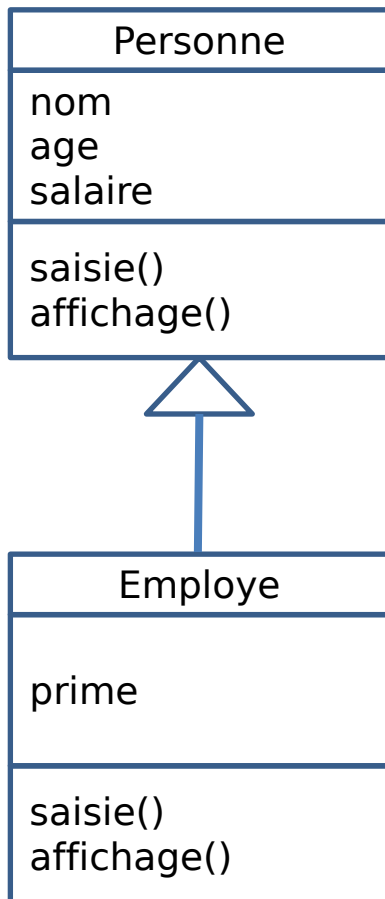
```
Nom : Pierre
Age : 25
Salaire : 1500
>> Affichage de p
Son nom est  Pierre
Son âge :   25
Son salaire : 1500.0
>> Affichage de q
Son nom est  Pierre
Son âge :   25
Son salaire : 1500.0
ref. identiques : False
```

← Le contenu des champs est le même
mais ce sont 2 références différentes.

Idée : L'héritage permet de construire une hiérarchie de classes. Les **classes héritières** héritent des champs et méthodes de la **classe ancêtre**.

➔ Ce mécanisme nécessite des efforts de modélisation et de conception.

Mais au final, on améliore la lisibilité et la réutilisabilité du code.



La classe **Employe** est une **Personne**, avec le champ supplémentaire **prime**. Cela va nécessiter la reprogrammation des méthodes **saisie()** et **affichage()**.

On peut éventuellement ajouter d'autres méthodes spécifiques à **Employe**.

Déclaration en Python

```
class Employe(Personne):  
    ...
```

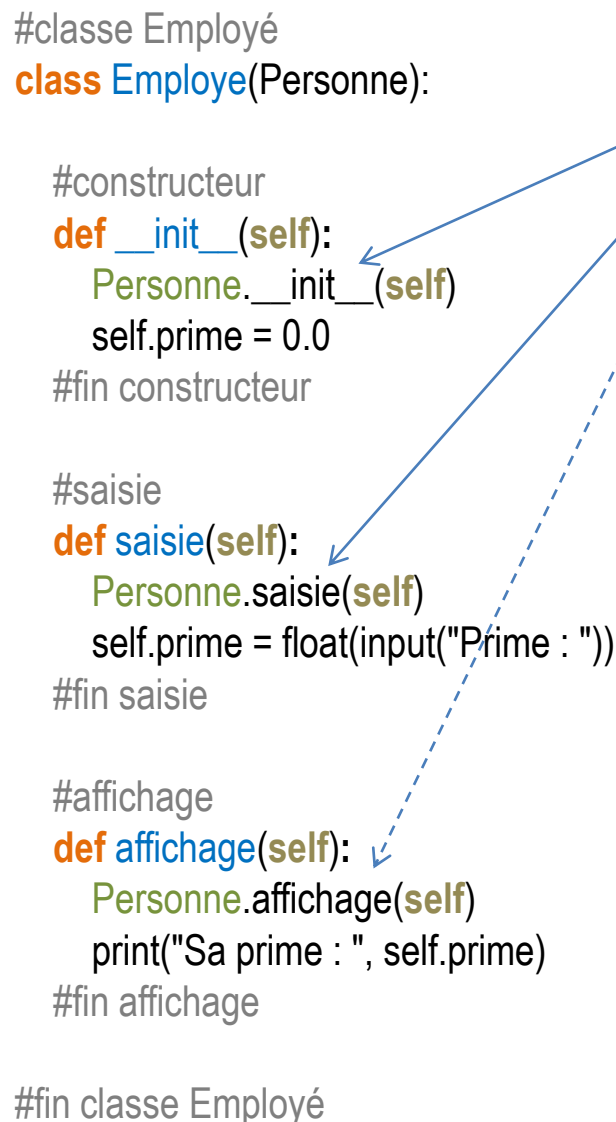
```
#classe Employé
class Employe(Personne):

    #constructeur
    def __init__(self):
        Personne.__init__(self)
        self.prime = 0.0
    #fin constructeur

    #saisie
    def saisie(self):
        Personne.saisie(self)
        self.prime = float(input("Prime : "))
    #fin saisie

    #affichage
    def affichage(self):
        Personne.affichage(self)
        print("Sa prime : ", self.prime)
    #fin affichage

#fin classe Employé
```



Surtout pas de copier/coller de code ! Noter comment sont **réutilisées** les méthodes programmées dans la classe ancêtre **Personne**.

Remarque : Noter l'utilisation de **self** dans ce contexte.

Le prog. principal ne présente aucune difficulté

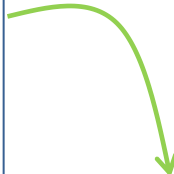
```
# -*- coding: utf -*-

#appel du module
import ModulePersonne as MP

#instanciation
e = MP.employe()

#saisie
e.saisie()

#affichage
print(">> Affichage")
e.affichage()
```



```
Nom : Zozo
Age : 36
Salaire : 1000
Prime : 500
>> Affichage
Son nom est  Zozo
Son âge :   36
Son salaire : 1000.0
Sa prime :  500.0
```


Liste polymorphe

Idée : Une collection peut contenir des objets de type différent. Cette caractéristique prend un sens particulier quand les objets sont issus de la même lignée.

Exemple d'exécution

```
Nb de pers : 2
1 Personne, 2 Employé : 1
1 Personne, 2 Employé : 2
*** début saisie
-----
Nom : Toto
Age : 36
Salaire : 1000
-----
Nom : Zaza
Age : 57
Salaire : 2500
Prime : 500
>>> début affichage
-----
Son nom est  Toto
Son âge :   36
Son salaire : 1000.0
-----
Son nom est  Zaza
Son âge :   57
Son salaire : 2500.0
Sa prime :   500.0
```

Selon la classe réellement instanciée (Employé ou Personne), les méthodes **saisie()** et **affichage()** adéquates seront appelées. C'est l'idée du polymorphisme.


```
# -*- coding: utf -*-
#appel du module
import ModulePersonne as MP
#liste vide
liste = []
#nb. de pers ?
n = int(input("Nb de pers : "))
#instanciation
for i in range(0,n):
    code = input("1 Personne, 2 Employé : ")
    if (code == "1"):
        m = MP.Personne()
    else:
        m = MP.Emloye()
    liste.append(m)
#saisie liste
print("*** début saisie")
for p in liste:
    print("-----")
    p.saisie()
#affichage
print(">>> début affichage")
for p in liste:
    print("-----")
    p.affichage()
```

Variable de classes

```
#début définition
class Personne:
    """Classe Personne"""


    #variable de classe
    compteur = 0

    #constructeur
    def __init__(self):
        #lister les champs
        self.nom = ""
        self.age = 0
        self.salaire = 0.0
        Personne.compteur += 1
    #fin constructeur
    ...
#fin définition
```



Noter que le champ est associé à la classe même (**Personne**) et non pas à l'instance (**self**).

Attention. Les affectations ne sont pas répercutées de la même manière selon qu'on la réalise sur la classe ou sur une instance. **A manipuler avec prudence !!!**



Une variable de classe est un champ directement accessible sur la classe, et qui est partagée par toutes les instances de la classe.

```
# -*- coding: utf -*-
#appel du module
import ModulePersonne as MP
#print compteur
print(MP.Personne.compteur)           → 0
#1ere instantiation
p = MP.Personne()
print(p.compteur)                     → 1
#2nde instantiation
q = MP.Personne()
print(q.compteur)                     → 2
#de nouveau compteur
print(MP.Personne.compteur)          → 2
```

```
#affectation 1
MP.Personne.compteur = 13
print(MP.Personne.compteur)           → 13
print(p.compteur)                     → 13
print(q.compteur)                     → 13
```

```
#affectation 2
p.compteur = 66
print(MP.Personne.compteur)           → 13
print(p.compteur)                     → 66
print(q.compteur)                     → 13
```