Chapitre 2 : Programmation Orientée Objet

Ahmed Ammar - IPEST 8 novembre 2020

Table des matières

1 Introduction

La Programmation Orientée Objet (POO) est une discipline de programmation dans laquelle le programmeur établit :

— non seulement les structures de données,

— mais aussi les opérations qui peuvent leurs être appliquées.

Ainsi,

— la structure de données devient un **objet** qui inclut

  — Données, appelées **attributs**

  — Opérations, appelées **méthodes**

— Le programmeur peut définir des **relations** entre les objets

2 Classe et Objet

2.1 Définitions

Une **classe** est équivalente à un nouveau type de données. On connaît déjà par exemple les classes `list` ou `str` et les nombreuses méthodes permettant de les manipuler, par exemple :

— `[3, 5, 1].sort()`

— `casse.upper()`

Un **objet** ou une **instance** est un exemplaire particulier d'une classe. Par exemple `[3, 5, 1]` est une instance de la classe `list` et `casse` est une instance de la classe `str`.

Les objets ont généralement deux sortes d'attributs : les données nommées simplement **attributs** et les fonctions applicables appelées **méthodes**.

Par exemple un objet de la classe `complex` possède :

— deux attributs : `imag` et `real` ;

— plusieurs méthodes, comme `conjugate()`, `abs()`...

La plupart des classes encapsulent à la fois les données et les méthodes applicables aux objets. Par exemple un objet `str` contient une chaîne de caractères Unicode (les données) et de nombreuses méthodes.

On peut définir un *objet* comme une *capsule* contenant des **attributs** et des **méthodes**.

2.2 Création de classe en Python

Une nouvelle classe est définie par le mot-clé `class`.

Syntaxe de la création :

```
class NomDeLaClasse :
    # Définition des attributs de la classe
    nom_attr = valeur # attribut de valeur commune pour toutes les
    ↪ instances
    ...
    # Définition des attributs d'objet (chaque instance a sa propre valeur)
```

```
    def __init__(self, parametres) : # le constructeur de l'objet
        self.nomattr1= v1
        self.nomattr2= v2
        ...

    # Définition des méthodes
        def nom_methode(self, autres_parametres) :
        # Corps de la méthode
```

Une classe permet de définir (déclarer) l'ensemble attributs et méthodes relatives à une catégorie d'objets.

- **Attributs de classe :** Un attribut de classe est défini au niveau de la classe et sa valeur est partagée par tous les objets instanciés de cette classe. L'accès à l'attribut est donné par : `NomDeLaClasse.nom_attribut`
- **Attributs d'objets :** Un attribut d'objet est défini au niveau de la méthode constructeur. La valeur d'un attribut d'objet est propre à chaque objet. L'accès à l'attribut est donné par : `nom_Objet.nom_attribut`
- **Le constructeur d'objet :** Le constructeur est une méthode particulière appelée lors de la création d'un nouvel objet permettant d'initialiser ses attributs. Le constructeur se définit dans une classe comme une fonction avec deux particularités :
  - le nom de la méthode doit être `__init__` ;
  - la méthode doit accepter au moins un paramètre de nom `self` qui apparaît en premier.
- **Le paramètre self :** Le paramètre `self` représente en fait l'objet cible, c'est-à-dire que c'est une variable qui référence l'objet en cours de création et permettant d'accéder à ses attributs et fonctionnalités.
- **Les méthodes :** Une méthode est une fonction qui agit principalement sur une instance de la classe. Elle doit accepter au moins le paramètre `self` figurant en première position. L'appel d'une méthode se fait par : `nom_Objet.nom_méthode(autres paramètres)`

2.3 Exemple : Une classe pour comptes bancaires

Le concept de compte bancaire dans un programme est un bon candidat pour un cours. Le compte comporte certaines données, généralement le nom du titulaire du compte, le numéro de compte et le solde courant. Trois choses que l'on peut faire avec un compte, c'est retirer de l'argent, mettre de l'argent sur le compte et imprimer les données du compte. Ces actions sont modélisées par des méthodes. Avec une classe, nous pouvons regrouper les données et les actions dans un nouveau type de données de sorte qu'un compte corresponde à une variable d'un programme.

Elle est créée ainsi :

```
# Création de la classe CompteBancaire
class CompteBancaire :
```

```
3     nomBanque = 'BIAT'  #Attributs de classe
4     def __init__(self, nom, num_compte, montant_initial) :
5     # Attributs d'objet
6         self.nom = nom
7         self.no = num_compte
8         self.sold = montant_initial
9
10    # Méthodes
11    def depot(self, montant) :
12        self.sold += montant
13
14    def retrait(self, montant) :
15        if self.sold >= montant :
16            self.sold -= montant
17        else :
18            raise Exception('retrait impossible')
19    def decharge(self) :
20        s = "{}, {}, solde  : {}".format(self.nom, self.no, self.sold)
21        print(s)
```

scripts/class.py

La classe `CompteBancaire` définie par :

- **Les attributs :**
  - Attributs de classe : `nomBanque` (nom de la banque) ;
  - Attributs d'objet : `no` (numéro du compte), `nom` (nom du propriétaire) et `sold` (solde).
- **Les méthodes :** `depot`, `decharge` et `retrait`.

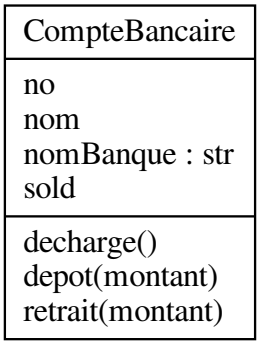

FIGURE 1 – Diagramme de classe `CompteBancaire`.

2.4 Création d'objets en Python

La création d'une **instance** (objet) d'une classe donnée se fait par un appel au nom de la classe avec les paramètres effectifs du constructeur, selon la syntaxe suivante :

```
>>> nom_obj = NomClasse(paramètres effectifs du constructeur)
```

Voici un test simple de la façon dont la classe `CompteBancaire` peut être utilisée :

```
>>> c1 = CompteBancaire('Mohamed Ahmed', '19371554951', 20000)
>>> c2 = CompteBancaire('Ali Tounsi',  '19371564761', 10000)
>>> c1.depot(1000)
>>> c1.retrait(4000)
>>> c2.retrait(8750)
>>> c1.retrait(1250)
>>> print("le solde de c1 : ", c1.sold)
le solde de c1 :  15750
>>> c1.decharge()
Mohamed Ahmed, 19371554951, solde : 15750
```

2.5 Constructeur

Les constructeurs sont généralement utilisés pour **instancier un objet**. La tâche des constructeurs consiste à initialiser (attribuer des valeurs) aux attributs de la classe lorsqu'un objet de la classe est créé.

En Python, la méthode `__init__()` est appelée le constructeur et est toujours appelée. quand un objet est créé.

Types de constructeurs :

— **Constructeur par défaut** : le constructeur par défaut est un constructeur simple qui naccepte aucun argument. Sa définition na quun seul argument qui soit une référence à linstance en cours de construction.

— **Constructeur paramétré** : Le constructeur paramétré prend son premier argument en tant que référence à l'instance en cours de construction, appelée self, et le reste des arguments est fourni par le programmeur.

En python, on ne peut définir qu'un seul constructeur :

```python
class Personne :

    # constructeur de la classe
    def __init__(self, nom, prenom) :
        self.nom = nom
        self.prenom = prenom

    def __del__(self) :
        print("je suis le destructeur")
```

scripts/Personne.py

2.6 Destructeur

Les destructeurs sont appelés lorsqu'un objet est détruit. En Python, les destructeurs ne sont pas aussi nécessaires que en d'autres langages de programmation, car Python dispose d'un **ramasse-miettes** qui gère automatiquement la gestion de la mémoire.

La méthode `__del __()` est une méthode appelée **destructeur** en Python. Il est appelé lorsque toutes les références à lobjet ont été supprimées, cest-à-dire lorsquun objet est nettoyé.

Exemple 1 : Classe Tableau :

Voici un exemple simple de destructeur :

```
1  class Tableau :
2      donnee=[]
3      # constructeur de la classe
4      def __init__(self) :
5          #initialiser le tableau avec 100 éléments
6          self.donnee=[0]*100
7
8      def __del__(self) :
9          print("je suis le destructeur")
10         # vider le tableau
11         self.donnee.clear()
```

scripts/Tableau.py

En utilisant le mot-clé `del`, nous avons supprimé toutes les références de lobjet `tab`. Le destructeur est donc invoqué automatiquement.

```
>>> tab = Tableau()
>>> del tab
je suis le destructeur
```

Note

Le destructeur est appeler à la fin du programme ou lorsque toutes les références à l'objet ont été supprimées. C'est-à-dire que le nombre de références devient zéro, et non lorsque l'objet est sorti de la portée.

Exemple 2 : Classe Personne :

Cet exemple donne une explication de la note.

```
1  class Personne :
2
3      # constructeur de la classe
4      def __init__(self, nom, prenom) :
5          self.nom = nom
6          self.prenom = prenom
```

```
7
8       def __del__(self) :
9           print("je suis le destructeur")
```

scripts/Personne.py

```
1  def creation() :
2      print("creation de l'objet")
3      p = Personne('TOUNSI','Mohamed')
4      print('fin de la création')
5      return p
6
7  print("Début du programme")
8  p1=creation()
9  print("Je suis {} {}, je suis un objet d'ID  : {}".format(p1.nom, p1.prenom,
       ↪  id(p1)))
10 print("fin du programme")
11 del p1
```

Notez que le destructeur est appelé après l'affichage de *"Fin du programme..."*

3 Méthodes spéciales et surcharge des opérateurs

Certaines méthodes de classe ont des noms qui commencent et se terminent par un double trait de soulignement. Ces méthodes permettent une syntaxe spéciale dans le programme et sont appelées **méthodes spéciales**. Le constructeur `__init__()` en est un exemple. Cette méthode est automatiquement appelée lorsqu'une instance est créée, mais nous n'avons pas besoin d'écrire explicitement `__init__()`. D'autres méthodes spéciales permettent d'effectuer des opérations arithmétiques avec des instances, de comparer des instances avec >, >=, !=, etc., d'appeler des instances comme nous appelons les fonctions ordinaires, et de tester si une instance évalue à Vrai ou Faux, pour mentionner certaines possibilités.

La **surcharge** permet à un opérateur de **posséder un sens différent** suivant le **type** de ses opérandes.

Par exemple, l'opérateur + permet :

```
1 x = 7 + 9 # addition entière
2 s = 'ab' + 'cd' # concaténation
```

Python possède des méthodes de surcharge pour :

— tous les types (`__call__`, `__str__` ...) ;

— les nombres (`__add__`, `__div__` ...) ;

— les séquences (`__len__`, `__iter__` ...).

3.1 Surcharge de l'appel fonctionnel

La méthode `__call__` permet aux programmeurs Python d'écrire des classes dont les instances se comportent comme des fonctions et peuvent être appelées comme une fonction. Lorsque l'instance est appelée comme une fonction ; si cette méthode est définie, `objet(arg1, arg2, ...)` est une abréviation de `objet.__call__(arg1, arg2, ...)`.

Exemple :

```
class Produit :
    def __init__(self) :
        print("Création de l'instance")

    # Définir la méthode __call__.
    def __call__(self, a, b) :
        print(a * b)

# Création de l'instance
ans = Produit()

# La méthode __call__ sera appelée
ans(10, 20)
```

scripts/Produit.py

3.2 Représentation formelle d'un objet

Pour afficher les informations relatives à un objet, en utilisant le nom de l'objet (représentation sur le shell) ou en utilisant la commande `print()` , il faut surcharger la méthode spéciale `__repr__` :

```
def __repr__(self) :
    return #la chaine qu'on veut afficher
```

Exemple :

```
class point :
    def __init__(self,a,b) :
        self.x=a
        self.y=b
    def __repr__(self) :
        return (str((self.x , self.y)))
```

```
>>> p=point(2,3)
>>> p # l'exécution de l'évaluation de p fait appel à __repr__
(2,3)
>>> print(p) # l'exécution de print fait appel à la méthode __repr__
(2,3)
```

3.3 Représentation informelle d'un objet

Pour donner une représentation textuelle informelle à un objet, il suffit de surcharger la méthode spéciale `__str__` :

```
def __str__(self) :
    return #la chaine qu'on veut afficher
```

Exemple 1 :

```
class point :
    def __init__(self,a,b) :
        self.x=a
        self.y=b
    def __str__(self) :
        return 'point' + str((self.x , self.y))
```

```
>>> p = point(2,3)
>>> p #sans redéfinir __repr__ l'exécution renvoie la référence de l'objet
<__main__.point object at 0x033DAB10>
>>> print(p) # l'exécution de print fait appel à la méthode __str__
point(2,3)
```

Exemple 2 :

```
class point :
    def __init__(self,a,b) :
        self.x=a
        self.y=b
    def __repr__(self) :
        return str((self.x , self.y))
    def __str__(self) :
        return 'point' + str((self.x , self.y))
```

```
>>> p = point(2,3)
>>> p # l'exécution de l'évaluation de p fait appel à __repr__
(2,3)
>>> print(p) # l'exécution de print fait appel à la méthode __str__
point(2,3)
```

3.4 Surcharge des opérateurs

La surcharge d'opérateurs permet la redéfinition et l'utilisation des opérateurs en fonction de la classe. Par exemple, l'utilisation de l'opérateur + pour additionner deux objets de même type.

Python associe à chaque opérateur une méthode spéciale qu'on peut surcharger, on cite dans la suite quelques exemples :

- Exemples des méthodes spéciales permettant la surcharge des opérateurs arithmétiques :
  - opérateurs unaires :
    - `+` : `__pos__(self)`
    - `-` : `__neg__(self)`
  - opérateurs binaires :
    - `+` : `__add__(self,other)`
    - `*` : `__mul__(self, other)`
    - `-` : `__sub__(self,other)`
    - ...
- Exemples des méthodes spéciales permettant la surcharge des opérateurs de comparaison :
  - `==` : `__eq__(self,other)`
  - `!=` : `__ne__(self, other)`
  - `>` : `__gt__(self,other)`
  - ...
- Exemples des méthodes spéciales permettant la surcharge des opérateurs d'indexation :
  - `objet[i]` : `__getitem__(self,indice)`
  - `objet[i] = v` : `__setitem__(self,indice,valeur)`

3.5 Exemple : Classe pour les vecteurs dans le plan

Cette partie explique comment implémenter des vecteurs bidimensionnels en Python de telle sorte que ces vecteurs agissent comme des objets que nous pouvons ajouter, soustraire, former des produits internes avec, et faire d'autres opérations mathématiques.

Les vecteurs dans le plan sont décrits par une paire de nombres réels, (a,b). Il existe des règles mathématiques pour ajouter et soustraire des vecteurs, multiplier deux vecteurs (le produit intérieur ou point ou scalaire), la longueur d'un vecteur, et la multiplication par un scalaire :

$$(a,b)+(c,d)=(a+c,b+d), \tag{1}$$

$$(a,b)-(c,d)=(a-c,b-d), \tag{2}$$

$$(a,b)\cdot(c,d)=ac+bd, \tag{3}$$

$$||(a,b)||=\sqrt{(a,b)\cdot(a,b)}. \tag{4}$$

De plus, deux vecteurs (a,b) et (c,d) sont égaux si $a=c$ et $b=d$.

Implémentation. Nous pouvons créer une classe pour les vecteurs de plan où les opérations mathématiques ci-dessus sont mises en uvre par des méthodes spéciales. La classe doit contenir deux attributs de données, un pour chaque composante du vecteur, appelés `x` et `y` ci-dessous. Nous incluons des méthodes spéciales pour l'addition, la soustraction, le produit scalaire (multiplication), la valeur absolue (longueur), la comparaison de deux vecteurs (== et !=), ainsi qu'une méthode pour imprimer un vecteur.

```
1  import math as m
2  class Vec2D(object) :
3      def __init__(self, x, y) :
4          self.x = x
5          self.y = y
6
7      def __add__(self, other) :
8          return Vec2D(self.x + other.x, self.y + other.y)
9
10     def __sub__(self, other) :
11         return Vec2D(self.x - other.x, self.y - other.y)
12
13     def __mul__(self, other) :
14         return self.x*other.x + self.y*other.y
15
16     def __abs__(self) :
17         return m.sqrt(self.x**2 + self.y**2)
18
19     def __eq__(self, other) :
20         return self.x == other.x and self.y == other.y
21
22     def __str__(self) :
23         return '({ :g}, { :g})'.format(self.x, self.y)
24
25     def __ne__(self, other) :
```

```
26          return not self.__eq__(other)  # réutiliser __eq__
```

scripts/Vec2D.py

Les méthodes `__add__`, `__sub__`, `__mul__`, `__abs__`, et `__eq__` devraient être assez simples à comprendre d'après les définitions mathématiques précédentes de ces opérations. La dernière méthode mérite un commentaire : ici, nous réutilisons simplement l'opérateur d'égalité `__eq__`, mais nous le faisons précéder d'un non. Nous aurions également pu mettre en oeuvre cette méthode comme :

```
1 def __ne__(self, autre) :
2     return self.x  != other.x or self.y  != other.y
```

Utilisation. Prenons quelques objets Vec2D :

```
>>> u = Vec2D(0,1)
>>> v = Vec2D(1,0)
>>> w = Vec2D(1,1)
>>> a = u + v
>>> print(a)
(1, 1)
>>> a == w
True
>>> a = u - v
>>> print(a)
(-1, 1)
>>> a = u*v
>>> print(a)
0
>>> print(abs(u))
1.0
>>> u == v
False
>>> u != v
True
```

Lorsque vous lisez cette présentation interactive, vous devez vérifier que le calcul est mathématiquement correct, que le type d'objet résultant d'un calcul est correct et que chaque calcul est effectué dans le programme. Ce dernier point est étudié en suivant le déroulement du programme à travers les méthodes de classe. À titre d'exemple, considérons l'expression `u != v`. Il s'agit d'une expression booléenne qui est `True` puisque u et v sont des vecteurs différents. Le type d'objet résultant doit être `bool`, avec des valeurs `True` ou `False`.

4 Héritage et polymorphisme

Un avantage décisif de la POO est qu'une classe Python peut toujours être spécialisée en une classe **fille** qui **hérite** alors de tous les attributs (données et méthodes) de sa **super classe** (classe mère). Comme tous les attributs peuvent être redéfinis, une méthode de la classe fille et de la classe mère peut posséder le même nom, mais effectuer des traitements différents (**surcharge**) et l'objet s'adaptera dynamiquement, dès l'instanciation. En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le **polymorphisme** permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode.

L'héritage se fait ainsi selon la syntaxe suivante :

```
class nom_sous_classe(nom_classe_mère) :
    #définir les attributs et les méthodes de la sous_classe
```

Note

L'héritage est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités supplémentaires ou différentes.

Le *polymorphisme* par *dérivation* est la faculté pour deux méthodes (ou plus) portant le même nom, mais appartenant à des classes héritées distinctes d'effectuer un travail différent. Cette propriété est acquise par la technique de la **surcharge**.

4.1 Exemple d'héritage et de polymorphisme

Dans l'exemple suivant, la classe `QuadrupedeDebout` hérite de la classe mère `Quadrupede`, et la méthode `piedsAuContactDuSol()` est polymorphe :

```
class Quadrupede :
    def piedsAuContactDuSol(self) :
        return 4

class QuadrupedeDebout(Quadrupede) :
    def piedsAuContactDuSol(self) :
        return 2
```

scripts/Quadrupede.py

Voici un test simple de la façon dont ces classes `Quadrupede` et `QuadrupedeDebout` peuvent être utilisées :

```
>>> chat = Quadrupede()
>>> chat.piedsAuContactDuSol()
```

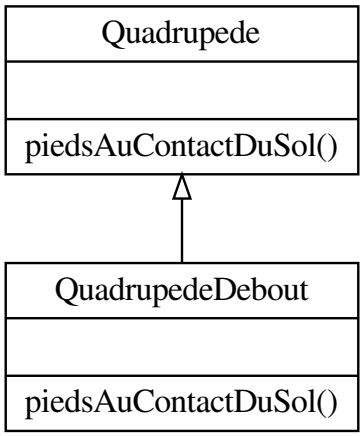

FIGURE 2 – Diagramme de classe `Quadrupede` et sa fille `QuadrupedeDebout`.

```
4
>>> homme = QuadrupedeDebout()
>>> homme.piedsAuContactDuSol()
2
```

4.2 Exemple d'héritage et de dérivation

La dérivation décrit la création de sous-classes par spécialisation. Elle repose sur la relation ń est-un ż.

On utilise dans ce cas le mécanisme de l'héritage.

L'implémentation Python utilisée est généralement l'appel à l'initialisateur de la classe parente dans l'initialisateur de la classe dérivée (utilisation de la fonction `super()`).

Dans l'exemple suivant, un **Carre** ń est-un ż **Rectangle** particulier pour lequel on appelle l'initialisateur de la classe mère avec les paramètres `longueur=cote` et `largeur=cote` :

```
1  class Rectangle :
2      def __init__(self, longueur=30, largeur=15) :
3          self.L, self.l = longueur, largeur
4          self.nom = "rectangle"
5      def __str__(self) :
6          return "nom  : {}".format(self.nom)
7
8  class Carre(Rectangle) : # héritage simple
9      """
10     Sous-classe spécialisée de la super-classe Rectangle.
11     """
12     def __init__(self, cote=20) :
```

```
13          # appel au constructeur de la super-classe de Carre :
14          super().__init__(cote, cote)
15          self.nom = "carré" # surcharge d'attribut
```

scripts/Rectangle.py

L'utilisation est comme suivant :

```
>>> r = Rectangle()
>>> c = Carre()
>>> print(r)
nom : rectangle
>>> print(c)
nom : carré
```

5 Application : Création des classes Pile et File

5.1 La classe Pile

Définition d'une pile. On rappelle qu'une pile est une structure de données qui suit le principe d'une pile d'assiettes, "*le dernier arrivé est le premier sorti*", on parle du mode **LIFO** (Last In First Out). L'insertion ou la suppression d'un élément ne peut se faire qu'à une seule extrémité, appelée sommet de la pile.

Une pile est définie par les opérations suivantes :

- **Empiler** : permet l'ajout d'un élément au sommet de la pile ;
- **Dépiler** : permet la suppression de l'élément au sommet de la pile si elle n'est pas vide ;
- **Vérifier** si une pile est vide ou non.

Implémentation d'une classe Pile. La classe Pile est définie par :

- L'attribut :
  - liste : initialisé par une liste vide
- Les méthodes :
  - `empiler` : permet l'ajout d'un élément donné à la fin de l'attribut `liste` ;
  - `depiler` : permet de supprimer et retourner le dernier élément de l'attribut `liste` s'il existe ;
  - `est_vide` : permet de vérifier si l'attribut `liste` est vide ou non.

```
1  class Pile :
2      def __init__(self) :
3          self.liste=[]
4      def empiler (self, v) :
5          self.liste.append(v)
6      def depiler(self) :
7          if self.est_vide() == False :
```

```
            return self.liste.pop()
    def est_vide(self)  :
        return self.liste == []
    def __repr__(self) :
        return (str(self.liste))
```

scripts/Pile.py

5.2 La classe File

Définition d'une File. On rappelle qu'une file est une structure de données qui suit le principe d'une file d'attente, "le premier arrivé est le premier sorti", on parle du mode **FIFO** (First In First Out).

Une file est définie par les opérations suivantes :

— **enfiler** : permet l'ajout d'un élément la fin de la file ;
— **défiler** : permet la suppression de l'élément au début de la file si elle n'est pas vide ;
— **vérifier** si une file est vide ou non.

Implémentation d'une classe File. La classe File est définie par :

— L'attribut :
  — liste : initialisé par une liste vide
— Les méthodes :
  — `enfiler` : permet l'ajout d'un élément donné à la fin de l'attribut `liste` ;
  — `defiler` : permet de supprimer et retourner le premier élément de lattribut `liste` sil existe ;
  — `est_vide` : permet de vérifier si l'attribut `liste` est vide ou non.

```
class File :
    def __init__(self)  :
        self.liste=[]
    def enfiler (self, v)  :
        self.liste.append(v)
    def defiler(self)  :
        if self.est_vide() == False :
              return self.liste.pop(0)
    def est_vide(self)  :
        return self.liste == []
    def __repr__(self) :
        return (str(self.liste))
```

scripts/File.py

5.3 Exemple d'inversion d'une pile

Il s'agit décrire une fonction `Inverser(p)` qui permet dinverser une pile `p` :

— avec utilisation d'une file intermédiaire ;

— sans utilisation d'une file intermédiaire.

```
# en utilisant une file intermédiaire
def Inverser(p) :
    f1 = File()
    while not p.est_vide() :
        f1.enfiler(p.depiler())
    while not f1.est_vide() :
        p.empiler(f1.defiler())
```

```
>>> p = Pile()
>>> p.empiler(2)
>>> p.empiler(6)
>>> p.empiler(8)
>>> p
[2, 6, 8]
>>> Inverser(p)
>>> p
[8, 6, 2]
```

```
# sans utiliser une file intermédiaire
def Inverser(p) :
    p1 = Pile()
    p2 = Pile()
    while not p.est_vide() :
        p1.empiler(p.depiler())
    while not p1.est_vide() :
        p2.empiler(p1.depiler())
    while not p2.est_vide() :
        p.empiler(p2.depiler())
```

```
>>> print(p)
[8, 6, 2]
>>> Inverser(p)
>>> print(p)
[2, 6, 8]
```