

Programmation Orientée Objet

Aide-Mémoire pour la préparation de l'épreuve
d'Informatique du Concours National d'Entrée aux
Ecoles d'Ingénieurs MP-PC-T 2020

Elaboré par :



Hayet Latrach (IPEIT)

Houcemeddine Filali (IPEIM)

Laila Hammami (IPEIN)

Wafa Meftah (IPEIS)

Mohamed Hammami (IPEIEM)

Besma Fayech (ENSIT)

Préambule

Ce document est destiné aux étudiants de deuxième année MP-PC-T des Instituts Préparatoires aux Etudes d'Ingénieurs.

Dans les circonstances exceptionnelles liées à la crise sanitaire mondiale et nationale du Covid19, un ensemble de supports de révision pour le programme d'informatique du concours national d'entrée aux écoles d'ingénieurs ont été élaborés essentiellement par les membres de la commission informatique du concours, suite à une action bénévole.

Le but de ces supports est d'accompagner les étudiants de tout le territoire Tunisien dans leur révision en leur fournissant à tous les mêmes documents, afin d'assurer une équité entre eux, qu'ils bénéficient ou non d'un enseignement ou tutorat à distance.

Toutefois, ces documents sont considérés comme des aide-mémoires et ne remplacent pas les cours assurés en présentiel.

Le document présent comporte les concepts de base associés à la programmation orientée objet en python. Il expose ainsi la représentation et la manipulation des classes, des objets, des méthodes, de l'héritage, du polymorphisme et de la surcharge. Une application intéressante aux piles et files est ensuite présentée. Des exercices d'application avec leurs corrigés sont également intégrés à la fin.

Table des matières

I	Introduction	3
II	Classe et Objet	3
II.1	Définitions.....	3
II.2	Création de classe en Python	4
II.3	Création d'objets en Python	5
III	Héritage / Dérivation	6
III.1	Définition	6
III.2	Exemple 1.....	6
III.3	Exemple 2.....	7
IV	Polymorphisme et surcharge	8
IV.1	Définitions.....	8
IV.2	Exemples.....	8
IV.3	Surcharge de méthodes spéciales.....	10
IV.3.1	Définition	10
IV.3.2	Représentation formelle d'un objet	10
IV.3.3	Représentation informelle d'un objet	10
IV.3.4	Surcharge des opérateurs.....	11
V	Application : Création des classes Pile et File.....	12
V.1	La classe Pile	12
V.1.1	Définition d'une pile.....	12
V.1.2	Implémentation d'une classe Pile.....	12
V.2	La classe File.....	12
V.2.1	Définition d'une File	12
V.2.2	Implémentation d'une classe File.....	13
V.3	Exemple d'inversion d'une pile	13
VI	Exercices d'application.....	14
VI.1	Exercice 1	14
VI.2	Exercice 2 : Extrait de l'examen SM-SP-IPEIT- Décembre 2018	14
VII	Alternatives de correction pour les exercices.....	16
VII.1	Corrigé Exercice 1	16
VII.2	Corrigé Exercice 2.....	17
VIII	Bibliographie et Netographie	20

I INTRODUCTION

En programmation procédurale, on s'intéresse aux étapes nécessaires pour résoudre un problème donné. Ces étapes sont modélisées par des tâches plus ou moins complexes, appelées procédures/fonctions qui manipulent les données du problème sans beaucoup de contrôle pour aboutir au résultat final. Le programme est donc un enchainement de ces tâches.

En programmation OO, on modélise le problème par un ensemble d'objets du monde réel qui interagissent entre eux à travers des messages. L'objet est donc l'entité principale.

L'objet encapsule ses propres données et les traitements relatifs à ses données dans une même entité. Le programme est un ensemble d'objets et des interactions entre ces objets.

II CLASSE ET OBJET

II.1 Définitions

Concrètement un objet est une entité du monde réel. Une personne, une voiture, une forme géométrique, une image, une feuille, etc.

En python, toutes les entités manipulées sont des objets (entier, fichier, module, etc)

La définition d'un objet (structure de l'objet) consiste à décrire ses propriétés :

- Statiques, appelées **attributs** ;
- Comportementales ou dynamiques, appelées **méthodes** :

Une feuille de papier par exemple, possède une couleur, une longueur et une largeur et on peut la plier, la déchirer, la couper, etc.

Un exemplaire d'un objet s'appelle une instance. Une feuille de papier de couleur rouge de longueur 21 cm et de largeur 17 cm est une instance de l'objet.

Deux instances possèdent chacune des valeurs d'attributs différentes mais se partagent les mêmes méthodes de l'objet. Il n'y a aucune interférence entre les objets d'une même classe.

Une classe est une entité qui permet la création des objets (instances) ayant les mêmes propriétés.

Pour pouvoir manipuler un objet (accéder à ses propriétés ou invoquer l'une de ses méthodes), on utilise l'opérateur point.

L'interaction entre les objets se fait par invocation des méthodes.

II.2 Création de classe en Python

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 : `nom_Classe.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, dont le nom doit être `self`, et qui doit être le premier paramètre.

- **Le paramètre `self`**

Le paramètre `self` représente en fait l'objet cible, c'est-à-dire que c'est une variable qui contient une référence vers l'objet qui est en cours de création. Grâce à ce dernier, on va pouvoir accéder aux attributs et fonctionnalités de l'objet cible.

- **Les méthodes**

Une méthode est une fonction qui agit principalement sur une instance de sa classe. Ainsi, une méthode doit accepter au moins le paramètre `self` qui doit toujours figurer en première position. L'appel d'une méthode se fait explicitement par :

`nom_Objet.nom_méthode (autres paramètres)`

- **Syntaxe de la création :**

```
class nom_classe:
    #définition des attributs de classe
        (attribut de valeur commune pour toutes les instances)
    nom_attr = valeur

    #définition des attributs d'objet (chaque instance a sa propre valeur)
    def __init__(self, liste_paramètres): # le constructeur de l'objet
        self.nomattr1= v1
        self.nomattr2= v2
        ...

    #définition des méthodes
    def nom_meth (self, autres_paramètres):
        #corps de la méthode
```

Exemple

Soit la classe Compte_bancaire définie par:

- **Les attributs:**
 - Attributs de classe : nombank
 - Attributs d'objet : NumeroCompte, NomProprietaire, solde
- **Les méthodes:** AfficherSolde, Retrait, Dépôt

Elle est créée ainsi :

```
# création de la classe Compte_Bancaire CompteBc
class compteBc:
    nombank= 'amenbank '
    def __init__(self, N, A, S) #la paramètre self désigne l'objet lui-même
        self.num=N
        self.nom=A
        self.sold=S
    def retrait(self,x):
        if self.sold>=x:
            self.sold= self.sold-x
        else:
            raise Exception('retrait impossible')
    def depot(self,x):
        self.sold+=x
    def AfficherSolde (self):
        print('compte au nom de ', self.nom, 'a pour solde',self.sold)
```

II.3 Création d'objets en Python

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

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

Exemple

Soit comptel une instance de la classe Compte_bancaire dont les données sont les suivantes :

- NumeroCompte : 512017
- NomProprietaire : mohamed
- solde : 1500

L'objet comptel est créé ainsi :

```
>>> comptel= compteBc(512017, mohamed, 1500)
>>> comptel.depot(500) # appel de la méthode depot
>>> comptel.AfficherSolde()
compte au nom de mohamed a pour solde 2000
```

III HERITAGE / DERIVATION

III.1 Définition

C'est le mécanisme qui permet de construire une classe « fille », ou sous-classe, à partir d'une classe « mère ». La classe fille ainsi obtenue hérite toutes les propriétés et toutes les fonctionnalités de son ancêtre. La classe « fille » peut éventuellement ajouter des méthodes/attributs propres à elle et redéfinir les méthodes qui ont été héritées de sa classe mère. La classe fille est une spécialisation de la classe mère.

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
```

III.2 Exemple 1

Nous proposons ci-après un exemple d'héritage entre la classe mère Point et la classe fille PointColoré :

```
>>> class Point:  
    def __init__(self,a,b):  
        self.x=a  
        self.y=b  
    def Affiche(self):  
        print('Point', (self.x, self.y))  
    def deplacer(self, dx, dy) :  
        self.x += dx  
        self.y += dy  
>>> class PointColoré(Point):  
    def __init__(self,a,b,col):  
        Point.__init__(self,a,b) # ou bien super().__init__(a,b)  
        self.couleur = col  
    def Affiche(self):  
        print('PointColoré', (self.x, self.y, self.couleur))  
  
>>> p2=PointColoré(2,3,'rouge'); p2.Affiche()  
PointColoré (2, 3, 'rouge')  
>>> p2.deplacer(1,1) #méthode héritée  
>>> p2.Affiche()  
PointColoré (3, 4, 'rouge')
```

III.3 Exemple 2

Comme exemple d'héritage à partir d'une classe native Python, nous proposons de définir une classe nommée `ListOfStr` permettant de stocker uniquement des valeurs de type `str`. La classe doit déclencher une exception pour toute tentative d'insertion/substitution d'un élément par une valeur qui n'est pas un `str`.

```
class ListOfStr(list):

    def __init__(self, source=()):
        """
        Construit une instance initialement vide
        """
        for item in source:
            if not isinstance(item, str):
                raise TypeError("type invalide !")
        super().__init__(source)

    def __repr__(self):
        return "ListOfStr(({ }))".format(super().__repr__())

    def append(self, other):
        if not isinstance(other, str):
            raise TypeError(" type invalide !")
        super().append(other)

    def extend(self, other):
        for item in other:
            if not isinstance(item, str):
                raise TypeError("type invalide !")
        super().extend(other)

    def __setitem__(self, index, item):
        if not isinstance(item, str):
            raise TypeError(" type invalide ! ")
        super().__setitem__(index, item)
```

```
>>> l = ListOfStr(("ali", "med", "zied"))
>>> l.append("zeineb")
>>> l
ListOfStr(['ali', 'med', 'zied', 'zeineb'])
>>> l[0] = "salah"
>>> l.extend(("hedi", "salah"))
>>> l.sort() ; l
ListOfStr(['hedi', 'med', 'salah', 'salah', 'zeineb', 'zied'])
```

IV POLYMORPHISME ET SURCHARGE

IV.1 Définitions

Le polymorphisme : est le concept permettant de redéfinir des méthodes de même nom avec des fonctionnalités similaires.

La surcharge est le mécanisme qui permet de redéfinir une méthode tout en gardant la même signature (c.à.d le nom de la méthode et la liste de ses paramètres).

IV.2 Exemples

Exemple 1

La classe Etudiant hérite de la classe Personne. La méthode **affiche()** est définie dans la classe mère et surchargée dans la classe fille :

```
>>> class Personne:
    def __init__(self,nom):
        self.nom=nom
    def affiche(self):
        print("je suis une personne")
>>> class Etudiant(Personne):
    def __init__(self,nom,cne):
        super().__init__(nom)
        self.cne=cne
    def affiche(self):
        print("je suis un étudiant")

>>> pr=Personne('Ali'); pr.affiche()
je suis une personne
>>> etd=Etudiant('Mohammed',123444); etd.affiche()
je suis un étudiant
```

Exemple 2

Il s'agit de calculer le périmètre et l'aire d'un carré, ainsi que le périmètre et l'aire d'un disque.

perimetre() et **aire()** sont deux méthodes polymorphes et surchargées dans les deux classes Carre et Disque :

```

>>> class Carre :
    " " " Un carré est modélisé par son coin supérieur gauche et une arête " " "
    def __init__ ( self, P, a ) :
        self.coin = P
        self.arete=a
    def perimetre ( self ) :
        return 4 * self.arete
    def aire ( self ) :
        return self.arete** 2

>>> class Disque :
    " " " Un disque est modélisé par son centre et son rayon " " "
    def __init__(self, c, r ) :
        self.centre = c
        self.rayon = r
    def perimetre ( self ) :
        return 2 * pi * self.rayon
    def aire ( self ) :
        return pi * self.rayon **2

>>> from math import pi

>>> p=Point(2,3); p1=Point(0,0);d=Disque(p,5);c=Carre(p1,2)
>>> d.perimetre(); c.perimetre()
31.41592653589793
8
>>> d.aire(); c.aire()
78.53981633974483
4

```

Exemple 3

L'opérateur + (la méthode `__add__`) est surchargé dans des différentes classes (float, complex, str,..) :

```

>>> x1 = 3 ; x2 = 5 ; x2 + x1    #float.__add__(x2, x1)
8
>>>z1 = 3+4j; z2 = 2-3j; z2 + z1 #complex.__add__(z2,z1)
5+1j
>>> s1 = « ali » ; s2 = « med » ; s2 + s1 #str.__add__(s2, s1)
"medali"
>>> t1 = (5,2); t2 = (3,); t2 +t1 # tuple.__add__(t2, t1)
(3, 5, 2)

```

IV.3 Surcharge de méthodes spéciales

IV.3.1 Définition

Une méthode spéciale est une méthode dont le nom est donné par `__nom__`. Elle est exécutée sans qu'on ait besoin de l'appeler explicitement. En fait, pour chaque méthode spéciale Python a un comportement par défaut.

Exemples :

`__init__` est appelée lors de la création d'une instance de classe par : `nom_obj=nomClasse(paramètres)`

`__str__` est appelée par la commande : `print(obj)`

`__add__` est appelée par : `obj1 + obj2`

IV.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)
```

IV.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

```

>>>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)

```

IV.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 numériques, ainsi que pour concaténer deux objets chaîne de caractères.

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)`

V APPLICATION : CREATION DES CLASSES PILE ET FILE

V.1 La classe Pile

V.1.1 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 servi*", 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.

V.1.2 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

```
class Pile :
    def __init__(self) :
        self.liste=[ ]
    def empiler (self, v) :
        self.liste.append(v)
    def depiler(self) :
        if self.est_vide( )== False:
            return self.liste.pop( )
    def est_vide( self) :
        return self.liste ==[ ]
```

V.2 La classe File

V.2.1 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 servi*", 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

V.2.2 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 l'attribut *liste* s'il 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 not self.est_vide( ) :
            return self.liste.pop(0 )
    def est_vide( self) :
        return self.liste ==[ ]
```

V.3 Exemple d'inversion d'une pile

Il s'agit d'écrire une fonction Inverser (p) qui permet d'inverser 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.defiler( ))
    while not f1.est_vide( ) :
        p.empiler( f1.defiler())
```

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

VI EXERCICES D'APPLICATION

VI.1 Exercice 1

On s'intéresse à ordonner les éléments d'une file d'entiers dans l'ordre croissant. On suppose définie la classe `file`, définir les méthodes suivantes :

1. `taille_file(F)` : qui permet de retourner la taille de la file `F`.
2. `Permutation_Circulaire(F,n)` : qui permet de réaliser `n` permutations circulaires à gauche.
3. `Inserer_Elt (F,elt)` qui permet d'insérer un élément dans une file `F` ordonnée selon l'ordre croissant .
4. `Ordonner_file(F)` qui permet, à partir d'une file `F`, de retourner une file `FO` ordonnée selon l'ordre croissant.

VI.2 Exercice 2 : Extrait de l'examen SM-SP-IPEIT- Décembre 2018

Dans ce problème, on s'intéresse à une application de gestion de stocks des différents magasins d'une enseigne. Un stock est composé par une collection d'articles. Un article admet une référence qui l'identifie, une désignation, et un prix d'achat. Un article est vendu avec un taux de bénéfice qui peut être modifié lors des promotions.

Afin de réaliser cette application, nous définissons une classe **Article**, une sous-classe **ArticleEnPromo** et une classe **Stock**.

A- Classe Article :

Définir la classe **Article** en écrivant :

1. Une méthode constructeur `__init__` qui permet d'initialiser les attributs **ref**, **dsg**, et **prix_a** de types respectivement entier, chaîne de caractères et réel. Ces attributs représentent les informations relatives à un article, à savoir, la référence, la désignation et le prix d'achat.
2. Une méthode de représentation `__repr__` qui permet d'afficher les différentes informations d'un article sous forme d'un tuple comprenant la référence, la désignation et le prix d'achat.
3. Une méthode **set_article** qui permet de modifier une des informations relatives à un article. Elle prend en paramètres :
 - `Self`.
 - **type_modif** de type entier pour identifier la modification: (1 : Modification de la référence, 2 : Modification de la désignation, 3 : Modification du prix d'achat).
 - **modif** qui représente l'information à modifier.

La méthode déclenche une exception intitulée `TypeError` si le type du paramètre **modif** ne coïncide pas avec le type de l'attribut concerné par la modification.

Indication: L'instruction : `raise nom_exception(msg)` déclenche l'exception `nom_exception` en affichant le message `msg`.

4. Une méthode **prix_vente** qui permet de calculer et retourner le prix de vente d'un article. Ce prix est calculé à partir du prix d'achat de l'article et d'un pourcentage **P** en appliquant la formule suivante :

$$\text{prix de vente} = \text{prix d'achat} + \text{prix d'achat} \times P.$$

Le pourcentage **P** est un paramètre de la méthode et admet par défaut la valeur 20%.

B- Classe ArticleEnPromo :

Définir la classe **ArticleEnPromo** qui hérite de la classe **Article** en écrivant :

1. Une méthode constructeur `__init__` qui permet de définir les mêmes attributs de la classe **Article** et d'initialiser l'attribut **remise** qui représente le taux de la remise accordée. Cette méthode prend en paramètre `self`, un article et le taux de remise.
2. Une méthode **prix_ventePromo** qui permet de retourner le prix de vente après la promotion.

C- Classe Stock :

Définir la classe **Stock** sachant que cette classe admet un attribut **LS** de type liste. Chaque élément de **LS** est une sous-liste qui comprend un article et la quantité en stock correspondante.

Pour cela, écrire:

1. Une méthode constructeur `__init__` qui permet d'initialiser l'attribut **LS** par une liste vide.
2. Une méthode de représentation `__repr__` qui permet d'afficher le contenu du stock. L'affichage sera fait ligne par ligne où chaque ligne est de la forme suivante :
(référence,désignation,prix_achat) : quantité en stock.
3. Une méthode **rechercher_article** qui permet, à partir de la référence, de chercher un article dans le stock. Elle retourne la désignation, le prix de vente et la quantité en stock de cet article s'il existe et `None` sinon.
4. Une méthode **ajouter_article** qui permet d'ajouter un nouvel article en stock.
5. Une méthode **maj_qtstock** qui permet, à partir de la référence d'un article et d'une quantité qt, de mettre à jour la quantité en stock d'un article existant. Cette méthode prend de plus un paramètre `type_maj` de type entier, il est égal à 1 dans le cas d'un ajout et -1 dans le cas d'une réduction.
6. Une méthode **supprimer_article** qui, à partir de la référence d'un article, permet de le supprimer.
7. Une méthode **articles_prix** qui permet de retourner la liste des articles dont le prix de vente est compris entre `prix1` et `prix2`.
8. Une méthode **articles_promo_qt** qui permet de créer et retourner une liste de tuples où chaque tuple comprend l'article qui sera en promotion ainsi que son prix de vente initial et son prix de vente après promotion.

Les articles qui seront en promotion sont ceux dont la quantité en stock est inférieure à une valeur `qt_min`. Le taux de remise et la valeur `qt_min` sont donnés en paramètres.

9. Soit **StockE** un dictionnaire qui contient les stocks de **n** magasins d'une enseigne. Les clés de **StockE** sont représentées par des entiers identifiant les différents magasins et les valeurs sont les stocks correspondants.
10. Ecrire un programme python qui permet de :
 - saisir une référence **ref** d'un article.
 - pour chaque magasin, afficher la quantité en stock de l'article identifié par la référence **ref**.
 - déterminer et afficher la quantité totale en stock de cet article.

VII ALTERNATIVES DE CORRECTION POUR LES EXERCICES

VII.1 Corrigé Exercice 1

```
class file:
    def __init__(self):
        self.l=[]
    def enfiler (self,v):
        self.l.append(v)
    def defiler(self):
        if not self.est_vide():
            return self.l.pop(0)
    def est_vide(self):
        return self.l==[]

# Question 1
def Taille(F):
    f1=file()
    n=0
    while not F.est_vide():
        f1.enfiler(F.defiler())
        n+=1
    while not f1.est_vide():
        F.enfiler(f1.defiler())
    return n

# Question 2
def permutation_circulaire(f,n):
    if not f.est_vide():
        for i in range(n):
            f.enfiler(f.defiler())

# Question 3
def Insérer_Elt(F,elt):
    if F.est_vide():
        F.enfiler(elt)

    else:
        n=Taille(F)
        ne=0
        for i in range(n):
            x=F.defiler()
            ne+=1
            if x<elt:
```

```

        F.enfiler(x)
    else:
        break
    F.enfiler(elt)
    if x>elt:
        F.enfiler(x)
    if n-ne>0:
        permutation_circulaire(F,n-ne)

# Question 4
def Ordonner_file(F):
    FO=file()

    while not F.est_vide():
        e=F.defiler()
        Inserer_Elt(FO,e)

    return FO

```

VII.2 Corrigé Exercice 2

```

class Article:
def __init__(self,ref,dsg,prix_a):
    self.ref=ref
    self.dsg=dsg
    self.prix_a=prix_a
def __repr__(self):
    return '('+str(self.ref)+','+self.dsg+', '+str(self.prix_a)+')'
def set_article(self,type_modif,modif,):
    if type_modif==1:
        if type(modif)!=int:
            raise ValueError('erreur de type')
        self.ref=modif
    elif type_modif==2:
        if type(modif)!=str:
            raise ValueError('erreur de type')
        self.dsg=modif
    elif type_modif==3:
        if type(modif)!=float:
            raise ValueError('erreur de type')
        self.prix_a=modif
def prix_vente(self,pourcentage=20):
    return self.prix_a*(1+pourcentage/100)

```

```
class ArticleEnPromo(Article):
    def __init__(self,A,remise):
        Article.__init__(self,A.ref,A.dsg,A.prix_a)
        self.remise=remise
    def prix_ventePromo(self):
        return self.prix_vente()*(1-self.remise/100)

class Stock:
    def __init__(self):
        self.LS=[]
    def __repr__(self):
        ch=''
        for i in self.LS:
            ch=ch+str(i[0])+':'+str(i[1])+'\n'
        return ch
    def recherche_article(self,ref):
        for i in self.LS:
            if i[0].ref==ref:
                return i[0].dsg,i[0].prix_vente(),i[1]
        return None
    def ajouter_article(self,A,qt):
        self.LS.append([A,qt])
    def maj_qtstock(self,ref,qt,type_maj):
        for i in self.LS:
            if i[0].ref==ref:
                if type_maj==1:
                    i[1]+=qt
                else:
                    i[1]-=qt
                break
    def supprimer_Articles(self,ref):
        for i in range(len(self.LS)):
            if self.LS[i][0].ref==ref:
                self.LS.pop(i)
    def articles_prix(self,p1,p2):
        L=[]
        for i in self.LS:
            pv=i[0].prix_vente()
```

```
        if pv>=p1 and pv<=p2:
            L.append(i[0])

    return L

def articles_promo_qt(self,qt_min,remise):
    L=[]
    for i in self.LS:
        if i[1]<qt_min:
            x=ArticleSolde(i[0],remise)
            L.append((x,x.prix_vente(),x.prix_venteSolde()))

    return L

#main
ref=int(input("donner le numéro du magasin: "))
qt_T=0
for m in StockE:
    for i in StockE[m].LS:
        if i[0].ref==ref:
            print(m,':',i[1])
            qt_T+=i[1]
```

VIII BIBLIOGRAPHIE ET NETOGRAPHIE

Ouvrages :

INFORMATIQUE Programmation et calcul scientifique en Python et Scilab, Thierry, Audibert & Amar Oussalah, Edition ellipses, 2014, ISBN: 978-2-7298-84710

Apprendre à programmer avec Python, Gérard Swinnen, Edition Eyrolles, ISBN : 978-2-212-13434-6

Supports de Cours

Support de Cours de H. Latrach POO, pour 2^{ème} année préparatoire MP-PC, IPEIT, Année universitaire 2019-2020.

Sites Web :

<https://courspython.com/classes-et-objets.html>

<https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python>

<https://developpement-informatique.com/article/152/polymorphisme-en-python>