

# Partie 1 : Partie théorique à faire sans ordinateur

## 1. Exercice 1 : Cryptage selon le « Code de César »

Notion abordée : Programmation objet.

Dans cet exercice, on étudie une méthode de chiffrement de chaînes de caractères alphabétiques. Pour des raisons historiques, cette méthode de chiffrement est appelée "code de César". On considère que les messages ne contiennent que les lettres capitales de l'alphabet "ABCDEFGHIJKLMNOPQRSTUVWXYZ" et la méthode de chiffrement utilise un nombre entier fixé appelé la clé de chiffrement.

Soit la classe **CodeCesar** définie ci-dessous :

```
class CodeCesar:
    def __init__(self, cle):
        self.cle = cle
        self.alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    def decale(self, lettre):
        num1 = self.alphabet.find(lettre)
        num2 = num1+self.cle
        if num2 >= 26:
            num2 = num2-26
        if num2 < 0:
            num2 = num2+26
        nouvelle_lettre = self.alphabet[num2]
        return nouvelle_lettre
```

On rappelle que la méthode **str.find(lettre)** renvoie l'indice (**index**) de la lettre dans la chaîne de caractères **str**

**Q1. Représenter** le résultat d'exécution du code Python suivant :

```
code1 = CodeCesar(3)
print(code1.decale('A'))
print(code1.decale('X'))
```

La méthode de chiffrement du « code César » consiste à décaler les lettres du message dans l'alphabet d'un nombre de rangs fixé par la clé. Par exemple, avec la clé 3, toutes les lettres sont décalées de 3 rangs vers la droite : le A devient le D, le B devient le E, etc.

**Q2. Ajouter** une méthode **cryptage(self, texte)** dans la classe **CodeCesar** définie à la question précédente, qui reçoit en paramètre une chaîne de caractères (le message à crypter) et qui retourne une chaîne de caractères (le message crypté). Cette méthode **cryptage(self, texte)** doit crypter la chaîne texte avec la clé de l'objet de la classe **CodeCesar** qui a été instancié.

### Exemple :

```
>>> code1 = CodeCesar(3)
>>> code1.cryptage("NSI")
'QVL'
```

### Q3. Ecrire un programme qui :

- (1) demande de saisir la clé de chiffrement
- (2) crée un objet de classe CodeCesar
- (3) demande de saisir le texte à chiffrer
- (4) affiche le texte chiffré en appelant la méthode cryptage

On ajoute la méthode **transforme(texte)** à la classe **CodeCesar** :

```
def transforme(self, texte):
    self.cle = -self.cle
    message = self.cryptage(texte)
    self.cle = -self.cle
    return message
```

On exécute la ligne suivante : **print(CodeCesar(10).transforme("PSX"))**

### Q4. Déterminer et justifier l'affichage obtenu par cette instruction.

## 2.Exercice 2

### Notion abordée : structures de données (dictionnaires)

Une ville souhaite gérer son parc de vélos en location partagée. L'ensemble de la flotte de vélos est stocké dans une table de données représentée en langage Python par un dictionnaire contenant des associations de type **id\_velo** : **dict\_velo** où **id\_velo** est un nombre entier compris entre 1 et 199 qui correspond à l'identifiant unique du vélo et **dict\_velo** est un dictionnaire dont les clés sont : **"type"**, **"etat"**, **"station"**.

Les valeurs associées aux clés **"type"**, **"etat"**, **"station"** de **dict\_velo** sont de type chaînes de caractères ou nombre entier :

**"type"** : chaîne de caractères qui peut prendre la valeur **"electrique"** ou **"classique"**

**"etat"** : nombre entier qui peut prendre la valeur 1 si le vélo est disponible, 0 si le vélo est en déplacement, -1 si le vélo est en panne

**"station"** : chaînes de caractères qui identifie la station où est garé le vélo.

Dans le cas où le vélo est en déplacement ou en panne, **"station"** correspond à celle où il a été dernièrement stationné.

Voici un extrait de la table de données, flotte étant une variable globale du programme :

```
flotte = {
12 : {"type" : "electrique", "etat" : 1, "station" : "Prefecture"},
80 : {"type" : "classique", "etat" : 0, "station" : "Saint-Leu"},
45 : {"type" : "classique", "etat" : 1, "station" : "Baraban"},
41 : {"type" : "classique", "etat" : -1, "station" : "Citadelle"},
26 : {"type" : "classique", "etat" : 1, "station" : "Coliseum"},
28 : {"type" : "electrique", "etat" : 0, "station" : "Coliseum"},
74 : {"type" : "electrique", "etat" : 1, "station" : "Jacobins"},
13 : {"type" : "classique", "etat" : 0, "station" : "Citadelle"},
83 : {"type" : "classique", "etat" : -1, "station" : "Saint-Leu"},
22 : {"type" : "electrique", "etat" : -1, "station" : "Joffre"}
}
```



Toutes les questions de cet exercice se réfèrent à l'extrait de la table flotte fourni ci-dessus.

**Q5. Déterminer** les valeurs renvoyées par les instructions suivantes :

```
x  flotte[26]
x  flotte[80]["etat"]
x  flotte[99]["etat"]
```

La fonction proposition est définie de la façon suivante :

```
def proposition(choix):
    for v in flotte:
        if flotte[v]["type"] == choix and flotte[v]["etat"] == 1:
            return flotte[v]["station"]
```

**Q6. Indiquer** les valeurs possibles de la variable **choix** puis **expliquer** ce que renvoie la fonction lorsque l'on choisit comme paramètre l'une des valeurs possibles de la variable **choix**.

**Q7. Écrire** un script en langage Python qui affiche les identifiants (**id\_velo**) de tous les vélos disponibles à la station "Citadelle".

**Q8. Écrire** un script en langage Python qui permet d'afficher l'identifiant (**id\_velo**) et la station de tous les vélos électriques qui ne sont pas en panne.

On dispose d'une table de données des positions GPS de toutes les stations, dont un extrait est donné ci-dessous. Cette table est stockée sous forme d'un dictionnaire. Chaque élément du dictionnaire est du type :

**'nom de la station' : (latitude, longitude)**

```
stations = {
'Prefecture' : (49.8905, 2.2967) ,
'Saint-Leu' : (49.8982, 2.3017),
'Coliseum' : (49.8942, 2.2874),
'Jacobins' : (49.8912, 2.3016)
}
```

On admet que l'on dispose d'une fonction **distance(p1, p2)** permettant de renvoyer la distance en mètres entre deux positions données par leurs coordonnées GPS (latitude et longitude).

Cette fonction prend en paramètre deux tuples représentant les coordonnées des deux positions GPS et renvoie un nombre entier représentant cette distance en mètres.

Par exemple, **distance((49.8905, 2.2967), (49.8912, 2.3016))** renvoie 359

**Q9. Écrire** une fonction qui prend en paramètre les coordonnées GPS de l'utilisateur sous forme d'un tuple et qui renvoie, pour chaque station située à moins de 800 mètres de l'utilisateur :

```
x  le nom de la station ;
x  la distance entre l'utilisateur et la station ;
x  les identifiants des vélos disponibles dans cette station.
```

*Remarque : Une station où aucun vélo n'est disponible ne doit pas être affichée.*

### 3.Exercice 3

Notion abordée : les arbres binaires de recherche.

Un arbre binaire est soit vide, soit un nœud qui a une valeur et au plus deux fils (le sous-arbre gauche et le sous-arbre droit).

- x x est un nœud, sa valeur est **x.valeur**
- x g1 est le fils gauche de X, noté **x.fils\_gauche**
- x d1 est le fils droit de x, noté **x.fils\_droit**

Un arbre binaire de recherche est ordonné de la manière suivante :

Pour chaque nœud X,

- les valeurs de tous les nœuds du sous-arbre gauche sont **strictement inférieures** à la valeur du nœud X
- les valeurs de tous les nœuds du sous-arbre droit sont **supérieures ou égales** à la valeur du nœud X

Ainsi, par exemple, toutes les valeurs des nœuds g1, g2 et g3 sont strictement inférieures à la valeur du nœud x et toutes les valeurs des nœuds d1, d2 et d3 sont supérieures ou égales à la valeur du nœud x.

Voici un exemple d'arbre binaire de recherche dans lequel on a stocké dans cet ordre les valeurs :

[26, 3, 42, 15, 29, 19, 13, 1, 32, 37, 30]

L'étiquette d'un nœud indique la valeur du nœud suivie du nom du nœud.

Les nœuds ont été nommés dans l'ordre de leur insertion dans l'arbre ci-dessous.

'29, noeud04' signifie que le nœud nommé **noeud04** possède la valeur 29.

On insère la valeur 25 dans l'arbre, dans un nouveau nœud nommé **noeud11**.

**Q10. Recopier** l'arbre binaire de recherche étudié et placer la valeur 25 sur cet arbre en coloriant en rouge le chemin parcouru. **Préciser** sous quel nœud la valeur 25 sera insérée et si elle est insérée en fils gauche ou en fils droit, et expliquer toutes les étapes de la décision.

**Q11. Préciser** toutes les valeurs entières que l'on peut stocker dans le nœud fils gauche du **noeud04** (vide pour l'instant), en respectant les règles sur les arbres binaires de recherche

Voici un algorithme récursif permettant de parcourir et d'afficher les valeurs de l'arbre :

```
fonction parcours(A) # A est un arbre binaire de recherche
    afficher(A.valeur)
    parcours(A.fils_gauche)
    parcours(A.fils_droit)
```

**Q12. Écrire** la liste de toutes les valeurs dans l'ordre où elles seront affichées.



## 4.Exercice 4


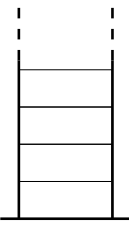
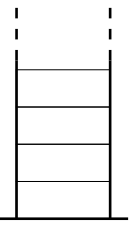
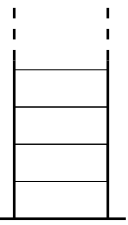

Notion abordée : structures de données : les piles.

Dans cet exercice, on considère une pile d'entiers positifs. On suppose que les quatre fonctions suivantes ont été programmées préalablement en langage Python :

- x `empiler(P, e)` : ajoute l'élément `e` sur la pile `P` ;
- x `depiler(P)` : enlève le sommet de la pile `P` et retourne la valeur de ce sommet ;
- x `est_vide(P)` : retourne `True` si la pile est vide et `False` sinon ;
- x `creer_pile()` : retourne une pile vide.

Dans cet exercice, seule l'utilisation de ces quatre fonctions sur la structure de données pile est autorisée.

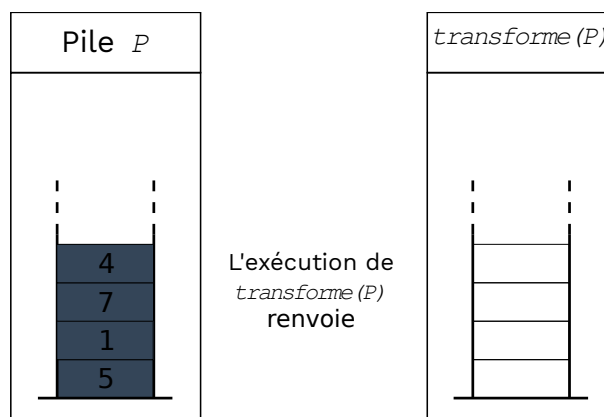
**Q13. Recopier** le schéma ci-dessous et le **compléter** sur votre copie en exécutant les appels de fonctions donnés. On écrira ce que renvoie la fonction utilisée dans chaque cas, et on indiquera **None** si la fonction ne retourne aucune valeur.

	Etape 0 <i>Pile d'origine P</i>	Etape 1 <i>empiler(P, 8)</i>	Etape 2 <i>depiler(P)</i>	Etape 3 <i>est_vide(P)</i>
				
Valeur renvoyée par la fonction				

On propose la fonction ci-dessous, qui prend en argument une pile `P` et renvoie un couple de piles :

```
def transforme(P) :
    Q = creer_pile()
    while not est_vide(P) :
        v = depiler(P)
        empiler(Q,v)
    return Q
```

**Q14. Recopier** et **compléter** sur votre copie le document ci-dessous



**Q15. Ecrire** une fonction en langage Python `maximum(P)` recevant une pile `P` comme argument et qui renvoie la valeur maximale de cette pile. On ne s'interdit pas qu'après exécution de la fonction, la pile soit vide.

On souhaite connaître le nombre d'éléments d'une pile à l'aide de la fonction `taille(P)`

**Q16. Proposer** une stratégie écrite en langage naturel et/ou expliquée à l'aide de schémas, qui permette de mettre en place une telle fonction. **Donner** le code Python de cette fonction `taille(P)` (on pourra utiliser les fonctions déjà programmées).

## Partie B : Travail sur ordinateur

### 5.Exercice 1 : Nombre d'occurrences dans un texte

L'occurrence d'un caractère dans un phrase est le nombre de fois où ce caractère est présent.

Exemples :

- l'occurrence du caractère 'o' dans 'bonjour' est 2 ;
- l'occurrence du caractère 'b' dans 'Bébé' est 1 ;
- l'occurrence du caractère 'B' dans 'Bébé' est 1 ;
- l'occurrence du caractère ' ' dans 'Hello world !' est 2.

On cherche les occurrences des caractères dans une phrase. On souhaite stocker ces occurrences dans un dictionnaire dont les clefs seraient les caractères de la phrase et les valeurs l'occurrence de ces caractères.

Par exemple : avec la phrase 'Hello world !' le dictionnaire est le suivant :

```
{ 'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 2, 'w': 1, 'r': 1, 'd': 1, '!': 1 }
```

(l'ordre des clefs n'ayant pas d'importance).

**Q17. Écrire** une fonction `occurrence_lettres` avec prenant comme paramètre une variable phrase de type `str`. Cette fonction doit renvoyer un dictionnaire de type constitué des occurrences des caractères présents dans la phrase.

### 6. Exercice 2 : Adressage IP

On définit une classe gérant une adresse IPv4.

On rappelle qu'une adresse IPv4 est une adresse de longueur 4 octets, notée en décimale à point, en séparant chacun des octets par un point. On considère un réseau privé avec une plage d'adresses IP de 192.168.0.0 à 192.168.0.255.

On considère que les adresses IP saisies sont valides.

Les adresses IP 192.168.0.0 et 192.168.0.255 sont des adresses réservées.



Le code ci-dessous implémente la classe **AdresseIP**.

```
class AdresseIP:
    def __init__(self, adresse):
        self.adresse = ...
    def liste_octet(self):
        """renvoie une liste de nombres entiers,
        la liste des octets de l'adresse IP"""
        return [int(i) for i in self.adresse.split(".")]

    def est_reservee(self):
        """renvoie True si l'adresse IP est une adresse
        réservée, False sinon"""
        return ... or ...

    def adresse_suivante(self):
        """renvoie un objet de AdresseIP avec l'adresse
        IP qui suit l'adresse self
        si elle existe et False sinon"""
        if ... < 254:
            octet_nouveau = ... + ...
            return AdresseIP('192.168.0.' + ...)
        else:
            return False
```

**Q18. Compléter** le code ci-dessus et instancier trois objets : **adresse1**, **adresse2**, **adresse3** avec respectivement les arguments suivants :

'192.168.0.1', '192.168.0.2', '192.168.0.0'

**Vérifier** que :

```
>>> adresse1.est_reservee()
False
>>> adresse3.est_reservee()
True
>>> adresse2.adresse_suivante().adresse
'192.168.0.3'
```