TP final – Bibliothèque de conversion de bases

Construire une bibliothèque de fonctions de conversion entre bases

NSI – Première – Séquence 3 : Les fonctions

Ce TP clôt la séquence sur les fonctions. Vous allez programmer les méthodes de conversion vues au chapitre 1 (division euclidienne successive, technique d'accumulation), puis les rassembler dans une véritable **bibliothèque** réutilisable, comme vu en cours.

Contrainte 1 : les listes ne sont pas encore au programme. Toutes vos fonctions devront donc se contenter de **chaînes de caractères**, construites petit à petit avec `+` – exactement comme `affichage_mot` au TP précédent. N'utilisez ni indexation (`chaine[i]`) ni découpage (`chaine[a:b]`).

Contrainte 2 (convention de tout le TP) : pour uniformiser l'écriture de la bibliothèque, **toutes** les fonctions de conversion prennent une chaîne de caractères en entrée et renvoient une chaîne de caractères en sortie – y compris pour un nombre écrit en base 10 ! Par exemple, l'écriture décimale de 57 sera toujours manipulée comme la chaîne `"57"`, jamais comme l'entier `57`.

Partie	Objectif	Durée indicative
A – Décimal / Binaire	Aller-retour décimal / binaire	30 min
B – Binaire / Hexadécimal	Regrouper par paquets de 4 bits	55 min
C – Combiner	Combiner, tester, empaqueter	35 min

1 Décimal et binaire : l'aller-retour

Remarque : Rappel de la méthode (chapitre 1)

Pour convertir un entier n en base b, on calcule le reste de la division euclidienne de n par b : c'est le premier chiffre obtenu (le chiffre de poids faible). On remplace ensuite n par le *quotient*, et on recommence, jusqu'à obtenir un quotient nul. On relit alors les restes obtenus **du dernier au premier**.
Exemple pour $n = 57$ et $b = 2$: $57 = 2\times 28+1$, $28 = 2\times 14+0$, $14 = 2\times 7+0$, $7 = 2\times 3+1$, $3 = 2\times 1+1$, $1 = 2\times 0+1$. En relisant les restes du dernier au premier : $57 = (111001)_2$.

Exercice 1: Écrire decimal_vers_binaire

On souhaite écrire une fonction `decimal_vers_binaire(chaine_dec)` qui calcule l'écriture binaire d'un entier naturel donné sous forme de chaîne.

- **Paramètre :** `chaine_dec`, une chaîne de caractères représentant un entier naturel en écriture décimale (par exemple `"57"`).
- **Valeur renvoyée :** une chaîne de caractères ne contenant que des `"0"` et des `"1"`, l'écriture binaire de ce nombre.
- **Cas particulier :** si le nombre vaut `0`, la fonction doit renvoyer directement `"0"`.

Commencez par convertir `chaine_dec` en entier grâce à `int(chaine_dec)`, puis réutilisez la méthode du rappel ci-dessus : tant que la valeur est `> 0`, calculez le chiffre (reste de la division par 2), placez-le **au début** de votre résultat (chaîne), comme vous l'avez fait pour `affichage_mot` au TP précédent, puis remplacez la valeur par le quotient.

Indication

`n = int(chaine_dec)` donne l'entier à convertir. Un chiffre `0` ou `1` (un entier) se convertit ensuite en caractère avec `str(...)`. Pensez à initialiser votre chaîne résultat à `""` avant la boucle `while`.

Exercice 2: Tester votre fonction

Écrire, à l'aide d'`assert`, un jeu de tests d'au moins trois cas pour `decimal_vers_binaire`, en incluant le cas particulier `chaine_dec = "0"` et un cas repris du rappel de cours (`"57"`).

Exercice 3: Comprendre l'accumulation (à la main)

Avant d'écrire le code de la fonction inverse, comprenons à la main la technique qui permet de convertir un nombre binaire en décimal, en lisant ses chiffres de **gauche à droite**.
Principe : on part d'une valeur égale à `0`. Pour chaque bit lu, on **double** la valeur courante, puis on lui **ajoute** ce bit.
Complétez le tableau suivant pour convertir $(1101)_2$ en décimal, en partant de `valeur = 0` :

bit lu	valeur avant	valeur × 2	+ bit = valeur après
1	0		
1			
0			
1			

Quelle est la valeur décimale finale obtenue ?

Exercice 4: Écrire binaire_vers_decimal

On souhaite maintenant écrire `binaire_vers_decimal(chaine_bin)` :

- **Paramètre :** `chaine_bin`, une chaîne de `"0"` / `"1"` (écriture binaire d'un nombre).
- **Valeur renvoyée :** une chaîne de caractères représentant la valeur décimale de ce nombre (rappel de la convention : toujours un `string` !).

Réutilisez exactement la technique tracée à la main à l'exercice précédent : parcourez `chaine_bin` avec un `for` (un bit à la fois), en accumulant la valeur au fur et à mesure, en partant de `valeur = 0`. N'oubliez pas de convertir le résultat final en chaîne avant de le renvoyer.

Exercice 5: Tester votre fonction

Écrire un jeu de tests pour `binaire_vers_decimal`, en incluant le cas `"0"` et un cas qui vérifie que l'on retrouve bien le nombre de départ de l'exercice 1 (`binaire_vers_decimal("111001")` doit redonner `"57"`).

2 Binaire et hexadécimal : l'aller-retour

Remarque : Rappel de la méthode (chapitre 1) – l'intérêt de l'hexadécimal

On regroupe le nombre binaire par paquets de **4 bits**, en partant du bit de poids faible (donc de la droite). Si le dernier paquet (celui de gauche) contient moins de 4 bits, on le complète par des `0` à gauche. On convertit ensuite chaque paquet en son symbole hexadécimal, à l'aide de la table de correspondance vue en cours (`0000` à `1111` correspondent à `0` à `F`).
Exemple : $(11\,1101)_2$ se regroupe en `0011` et `1101`, soit `3` et `D` : $(11\,1101)_2 = (3D)_{16}$.
C'est précisément pour cela que l'hexadécimal est si pratique en informatique : **un seul symbole hexadécimal résume exactement 4 bits**, ce qui rend l'écriture binaire beaucoup plus compacte et lisible (un octet, 8 bits, s'écrit toujours avec exactement 2 symboles hexadécimaux).

Exercice 6: Chiffre et symbole hexadécimal

Les chiffres hexadécimaux supérieurs à 9 s'écrivent avec des lettres : `10` → `A`, `11` → `B`, `12` → `C`, `13` → `D`, `14` → `E`, `15` → `F`.
Écrire deux fonctions, à l'aide de `if` / `elif` :

1. `chiffre_vers_symbole(chiffre)` : paramètre `chiffre`, un entier entre 0 et 15 ; renvoie le symbole correspondant (une chaîne d'un seul caractère : `"0"` à `"9"`, ou `"A"` à `"F"`).
2. `symbole_vers_chiffre(caractere)` : la fonction inverse – paramètre `caractere`, un symbole (`"0"` à `"9"` ou `"A"` à `"F"`) ; renvoie l'entier correspondant (entre 0 et 15).

Indication

Pour `chiffre_vers_symbole`, si `chiffre < 10`, `str(chiffre)` suffit à obtenir directement le bon caractère. Pour `symbole_vers_chiffre`, le test `caractere in "0123456789"` permet de repérer les chiffres usuels, et `int(caractere)` les convertit alors directement en entier.

Exercice 7: Valeur d'un paquet de 4 bits

On souhaite écrire une fonction `valeur_groupe(groupe)` :

- **Paramètre :** `groupe`, une chaîne de **exactement 4** caractères `"0"` / `"1"` (un paquet de bits).
- **Valeur renvoyée :** un entier entre 0 et 15, la valeur décimale de ce paquet.

Écrire cette fonction en parcourant `groupe` avec un `for` (un caractère à la fois), et en accumulant la valeur au fur et à mesure – exactement la même technique qu'à l'exercice de trace de la partie A, appliquée ici à 4 bits seulement.

Indication

`int(bit)` convertit le caractère `"0"` ou `"1"` en entier. La mise à jour à chaque tour de boucle s'écrit `valeur = valeur * 2 + int(bit)`, en partant de `valeur = 0`.

Exercice 8: Code à trous : binaire_vers_hexa

On souhaite maintenant écrire `binaire_vers_hexa(chaine_bin)`, qui renvoie l'écriture hexadécimale du nombre binaire `chaine_bin`, en suivant la méthode du rappel de cours : compléter à gauche par des `"0"` jusqu'à obtenir un multiple de 4 caractères, puis convertir chaque paquet de 4 bits en un symbole hexadécimal. Voici le squelette de cette fonction, avec deux emplacements manquants, numérotés `(1)` et `(2)` :

```
def binaire_vers_hexa(chaine_bin):
    """
    Précondition : chaine_bin est une chaîne de "0"/"1".
    Renvoie l'écriture hexadécimale du même nombre, sous forme de chaîne.
    """
    while len(chaine_bin) % 4 != 0:
        chaine_bin = "0" + chaine_bin

    resultat = ""
    groupe = ""
    for bit in chaine_bin:
        groupe = groupe + bit
        if len(groupe) == 4:
            resultat = resultat + (1)
            groupe = (2)
    return resultat
```

- `(1)` doit convertir le paquet `groupe` en son symbole hexadécimal (deux fonctions à combiner : une de cet exercice, une de l'exercice « Valeur d'un paquet de 4 bits ») ;
- `(2)` doit réinitialiser `groupe` pour préparer le paquet suivant.

Recopiez la fonction complète avec les deux emplacements remplis.

Exercice 9: Un chiffre hexadécimal sur exactement 4 bits

Pour convertir dans l'autre sens (hexadécimal vers binaire), il nous faut l'opération inverse de `valeur_groupe` : transformer un chiffre (0 à 15) en un paquet de **exactement 4** bits, quitte à ajouter des `"0"` devant. Écrire `chiffre_vers_binaire4(chiffre)` :

- **Paramètre :** `chiffre`, un entier entre 0 et 15.
- **Valeur renvoyée :** une chaîne de **exactement 4** caractères `"0"` / `"1"`, l'écriture binaire de `chiffre`, complétée par des `"0"` à gauche si besoin.

Ne réécrivez pas l'algorithme de conversion : réutilisez `decimal_vers_binaire` (partie A) pour obtenir l'écriture binaire de `chiffre`, puis complétez-la à gauche par des `"0"` jusqu'à atteindre 4 caractères (même technique que dans `binaire_vers_hexa`).

Exercice 10: Écrire hexa_vers_binaire

On peut maintenant écrire `hexa_vers_binaire(chaine_hexa)` : convertir chaque symbole hexadécimal en son paquet de 4 bits (exercice précédent), puis les mettre bout à bout.
On vous fournit une fonction utilitaire, `enlever_zeros_inutiles`, qui retire les `"0"` inutiles au début d'une chaîne binaire (par exemple, pour que le résultat s'écrive `"111101"` et non `"00111101"`) :

```
def enlever_zeros_inutiles(chaine):
    """
```

```
    Renvoie chaine privée de ses "0" de tête (mais renvoie "0" si chaine
    ne contient que des "0").
    """
    resultat = ""
    a_commence = False
    for caractere in chaine:
        if caractere == "1":
            a_commence = True
        if a_commence:
            resultat = resultat + caractere
    if resultat == "":
        resultat = "0"
    return resultat
```

À vous d'écrire `hexa_vers_binaire(chaine_hexa)` : parcourez `chaine_hexa` avec un `for`, convertissez chaque caractère en son paquet de 4 bits et mettez-les bout à bout (avec `+`), puis appelez `enlever_zeros_inutiles` sur le résultat avant de le renvoyer.

Exercice 11: Un jeu de tests

Écrire un jeu de tests pour `valeur_groupe`, `binaire_vers_hexa`, `chiffre_vers_binaire4` et `hexa_vers_binaire` (au moins deux cas par fonction), en pensant au cas particulier `"0"`.

3 Combiner pour hexadécimal et décimal

Il vous manque encore deux fonctions : la conversion directe du décimal vers l'hexadécimal, et de l'hexadécimal vers le décimal. Bonne nouvelle : le **binaire fait le pont** entre les deux, et vous avez déjà tout ce qu'il faut.

Exercice 12: Combiner pour decimal_vers_hexa et hexa_vers_decimal

En réutilisant **uniquement** des fonctions déjà écrites (aucune nouvelle boucle, aucun nouveau calcul), écrire :

- `decimal_vers_hexa(chaine_dec)` : passe par le binaire (`decimal_vers_binaire`, puis `binaire_vers_hexa`).
- `hexa_vers_decimal(chaine_hexa)` : passe par le binaire (`hexa_vers_binaire`, puis `binaire_vers_decimal`).

Chacune de ces deux fonctions tient en deux lignes.

Exercice 13: Un jeu de tests complet

Écrire un jeu de tests couvrant les six fonctions de la bibliothèque (`decimal_vers_binaire`, `binaire_vers_decimal`, `hexa_vers_binaire`, `binaire_vers_hexa`, `decimal_vers_hexa`, `hexa_vers_decimal`), en pensant au cas particulier `"0"` et à au moins un aller-retour (par exemple, vérifier que convertir un nombre puis reconvertir le résultat redonne le nombre de départ).

Exercice 14: En faire une vraie bibliothèque

1. Enregistrer toutes vos fonctions (avec leurs docstrings) dans un fichier nommé `conversions.py`.
2. Dans un **nouveau** fichier, situé dans le même dossier (par exemple `programme_principal.py`), importer votre bibliothèque et l'utiliser pour convertir trois nombres de votre choix : un du décimal vers l'hexadécimal, un de l'hexadécimal vers le décimal, et un du décimal vers le binaire.

Remarque : Ce qu'il faut retenir de ce TP

Vous avez transformé des méthodes de conversion faites « à la main » au chapitre 2 en une bibliothèque de fonctions testées et réutilisables. Le **binaire a servi de pivot** entre le décimal et l'hexadécimal : plutôt que de réinventer un algorithme pour chaque nouvelle conversion, vous avez **combiné** des fonctions déjà écrites et déjà testées. C'est exactement l'intérêt d'une bonne décomposition en fonctions – et c'est aussi pour cela qu'on a adopté, dans tout ce TP, la même convention d'entrée/sortie (toujours une chaîne de caractères) pour chaque fonction de la bibliothèque.