Les entiers et leurs représentations

Chap. 14,02

Préliminaire. Lors de cette séance toutes les fonctions à programmer sont données avec leur spécification. Cette dernière comporte des tests et leurs résultats. Je vous rappelle que pour exécuter automatiquement ces tests les dernière lignes du fichier contenant votre code doivent se conclure par :

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

1 Les entiers naturels

1.1 La représentation binaire

1) Définir la fonction `octet` dont la spécification est :

```
def octet(dec: str) -> str:
    """
    Vérifie si la chaîne contenant la représentation d'un
    nombre entier n < 256 en base 2 est agencée sous forme
    d'un octet.
    Ajoute des 0 si nécessaire.

    >>> octet('1101')
    '00001101'
    >>> octet('11010000')
    '11010000'
    """
```

▶ **Solution :**

```
def octet(dec: str) -> str:
    """
    Vérifie si la chaîne contenant la représentation d'un nombre entier n < 256
    en base 2 est agencée sous forme d'un octet.
    Ajoute des 0 si nécessaire.

    >>> octet('1101')
    '00001101'
    >>> octet('11010000')
    '11010000'
    """
    lg_octet = 8
    manque = lg_octet - len(dec)

    if manque > 0:
```

```
        for i in range(manque):
            dec = str(0) + dec

    return dec
```

2) Définir la fonction `decompose` dont la spécification est :

```
def decompose(n: int) -> str:
    """
    Représentation du nombre n dans la base 2. La grandeur
    retournée est une chaine de caractères représentant le
    résultat de la décomposition, les valeurs de poids le
    plus élevé en premier sous forme d'un octet.

    HYPOTHÈSE : n < 256
    REMARQUE : decompose utilise octet

    >>> decompose(13)
    '00001101'
    >>> decompose(0)
    '00000000'
    """
```

▶ **Solution :**

```
def decompose(n: int) -> str:
    """
    Représentation du nombre n dans la base 2. La grandeur retournée est
    une chaine de caractères représentant le résultat de la décomposition, les
    valeurs de poids le plus élevé en premier sous forme d'un octet.

    HYPOTHÈSE : n < 256
    REMARQUE : decompose utilise octet

    >>> decompose(13)
    '00001101'
    >>> decompose(0)
    '00000000'
    """
    res = ""
    if n == 0:
        res = '0'
    else:
        while n != 0:
            # ajout du reste de la division euclidienne au début de la liste
            res = str(n % 2) + res
            # poursuite la décomposition avec le quotient de la division
            n = n // 2
    return octet(res)
```

3) Définir la fonction `recompose` dont la spécification est :

```
def recompose(dec: str) -> int:
    """
    Détermine la valeur du nombre représenté par dec dans la base
    2. La chaîne doit être organisée de telle sorte que les
    valeurs de poids le plus élevé apparaissent en premier.

    HYPOTHÈSE : la chaîne est binaire et est codée sur un octet.

    >>> recompose('11010100')
    212
    >>> recompose('00000000')
    0
    """
```

▶ **Solution :**

```
def recompose(dec: str) -> int:
    """
    Détermine la valeur du nombre représenté par dec dans la base 2.
    La chaîne doit être organisée de telle sorte que les valeurs de poids le
    plus élevé apparaissent en premier.

    HYPOTHÈSE : la chaîne est binaire et est codée sur un octet.

    >>> recompose('11010100')
    212
    >>> recompose('00000000')
    0
    """
    nbre = 0
    for i in range(len(dec)):
        nbre = nbre + int(dec[i]) * 2**(len(dec) - 1 - i)
    return nbre
```

1.2 La représentation hexadécimale

4) Définir la fonction `decompose_16` dont la spécification (et la première ligne de code) est :

```
def decompose_16(n: int) -> List[int]:
    """
    Représentation du nombre n dans la base 16. La grandeur retournée est
    une chaîne représentant le résultat de la décomposition, les valeurs de poids
    le plus élevé en premier.

    REMARQUE : La fonction appelle les fonctions decompose et octet.

    >>> decompose_16(13)
    '0xD'
    >>> decompose_16(0)
    '0x0'
```

```
    >>> decompose_16(347)
    '0x15B'
    """
    ref = {10: 'A', 11: 'B', 12: 'C', 13: 'D', 14: 'E', 15: 'F'}
```

▶ **Solution :**

```
def decompose_16(n: int) -> List[int]:
    """
    Représentation du nombre n dans la base 16. La grandeur retournée est
    une chaîne représentant le résultat de la décomposition, les valeurs de poids
    le plus élevé en premier.

    La fonction appelle les fonctions decompose et octet.

    >>> decompose_16(13)
    '0xD'
    >>> decompose_16(0)
    '0x0'
    >>> decompose_16(347)
    '0x15B'
    """
    ref = {10: 'A', 11: 'B', 12: 'C', 13: 'D', 14: 'E', 15: 'F'}

    res = ""
    if n == 0:
        res = '0'
    else:
        while n != 0:
            reste = n % 16
            if reste < 10:
                res = str(reste) + res
            else:
                res = ref[reste] + res
            n = n // 16
    return '0x' + res
```

5) Définir la fonction `recompose_16` dont la spécification est :

```
def recompose_16(dec: str) -> int:
    """
    Détermine la valeur du nombre représenté par dec dans la base 16.
    La chaîne doit être organisée de telle sorte que les valeurs de poids le
    plus élevé apparaissent en premier.

    >>> recompose_16('0x15')
    21
    >>> recompose_16('0x15B')
    347
    >>> recompose_16('0xF')
    15
```

```
    """
    ref = {'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15}
```

▶ **Solution :**

```
def recompose_16(dec: str) -> int:
    """
    Détermine la valeur du nombre représenté par dec dans la base 16.
    La chaîne doit être organisée de telle sorte que les valeurs de poids le
    plus élevé apparaissent en premier.

    >>> recompose_16('0x15')
    21
    >>> recompose_16('0x15B')
    347
    >>> recompose_16('0xF')
    15
    """
    ref = {'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15}

    nbre = 0
    chaine = dec[2:]  # On élimine les caractères '0x'
    for i in range(len(chaine)):
        car = chaine[i]  # Caractère représentant un nombre
        if car in ref.keys():
            nbre = nbre + ref[car] * 16**(len(chaine) - 1 - i)
        else:
            nbre = nbre + int(car) * 16**(len(chaine) - 1 - i)
    return nbre
```

6) Définir la foncton `de_bin_vers_hex` dont la spécification est :

```
def de_bin_vers_hex(dec: str) -> str:
    """
    À partir de dec, la représentation binaire d'un nombre entier, détermination
    de sa représentation hexadécimale en regroupant les bits par 4.

    HYPOTHÈSE : la chaîne binaire est un octet.
    REMARQUE : de_bin_vers_hex doit utiliser recompose et decompose_16.

    >>> de_bin_vers_hex('00001101')
    '0D'
    >>> de_bin_vers_hex('11111111')
    'FF'
    """
```

▶ **Solution :**

```
def de_bin_vers_hex(dec: str) -> str:
    """
```

```
    À partir de dec, la représentation binaire d'un nombre entier, détermination
    de sa représentation hexadécimale en regroupant les bits par 4.

    HYPOTHÈSE : la chaîne binaire est un octet.
    REMARQUE : de_bin_vers_hex doit utiliser recompose et decompose_16.

    >>> de_bin_vers_hex('00001101')
    '0D'
    >>> de_bin_vers_hex('11111111')
    'FF'
    """
    res_hex = ""

    dec = [dec[:4], dec[4:]]
    for elt in dec:
        nbre = recompose(elt)
        dec_hex = decompose_16(nbre)
        dec_hex = dec_hex[2:]  # On élimine les deux premiers caractères 0x
        res_hex = res_hex + dec_hex
    return res_hex
```

7) Définir la fonction `de_hex_vers_bin` dont la spécification est :

```
def de_hex_vers_bin(dec: str) -> str:
    """
    À partir de dec, la représentation hexadécimale d'un nombre entier, détermination
    de sa représentation binaire.

    HYPOTHÈSE : la chaîne binaire est un octet.
    REMARQUE : de_bin_vers_hex doit utiliser recompose et decompose_16.

    >>> de_hex_vers_bin('0D')
    '00001101'
    >>> de_hex_vers_bin('FF')
    '11111111'
    """
```

▶ **Solution :**

```
def de_hex_vers_bin(dec: str) -> str:
    """
    À partir de dec, la représentation hexadécimale d'un nombre entier, détermination
    de sa représentation binaire.

    HYPOTHÈSE : la chaîne binaire est un octet.
    REMARQUE : de_bin_vers_hex doit utiliser recompose et decompose_16.

    >>> de_hex_vers_bin('0D')
    '00001101'
    >>> de_hex_vers_bin('FF')
    '11111111'
```

```
    """
    res_bin = ""

    for elt in dec:
        nbre = recompose_16('0x' + elt)  # Ajout de 0x nécessaire car recompose
l'attend
        dec_bin = decompose(nbre)
        dec_bin = dec_bin[4:]  # On retire les 4 premiers 0 qui ne servent à rien ici

        res_bin = res_bin + dec_bin

    return res_bin
```

2 Les entiers relatifs

8) Définir la fonction `decompose_relatif` dont la spécification est :

```
def decompose_relatif(n: int) -> str:
    """
    Retourne une chaîne de caractères qui donne la représentation d'un entier relatif
    en base 2 en utilisant le complément à 2.

    HYPOTHÈSE : n peut être codé sur un octet.
    REMARQUE : decompose_relatif utilise decompose.

    >>> decompose_relatif(127)
    '01111111'
    >>> decompose_relatif(0)
    '00000000'
    >>> decompose_relatif(-1)
    '11111111'
    >>> decompose_relatif(-128)
    '10000000'
    """
```

▶ **Solution :**

```
def decompose_relatif(n: int) -> str:
    """
    Retourne une chaîne de caractères qui donne la représentation d'un entier relatif
    en base 2 en utilisant le complément à 2.

    HYPOTHÈSE : -128 <= n <= 127
    REMARQUE : decompose_relatif utilise decompose.

    >>> decompose_relatif(127)
    '01111111'
    >>> decompose_relatif(0)
    '00000000'
    >>> decompose_relatif(-1)
```

```
    '11111111'
    >>> decompose_relatif(-128)
    '10000000'
    """
    lg_octet = 8

    if n < 0 :
        return decompose(n + 2**lg_octet)
    else:
        return decompose(n)
```

9) Définir la fonction `recompose_relatif` dont la spécification est :

```
def recompose_relatif(dec: str) -> int:
    """
    Détermine la valeur de l'entier relatif représenté par dec dans la base 2
    lorsqu'on utilise le complément à 2.
    La chaîne doit être organisée de telle sorte que les valeurs de poids le
    plus élevé apparaissent en premier.

    HYPOTHÈSE : la chaîne est binaire et est codée sur un octet.
    REMARQUE : recompose_relatif utilise recompose.

    >>> recompose_relatif('10000000')
    -128
    >>> recompose_relatif('01111111')
    127
    >>> recompose_relatif('00000000')
    0
    """
```

► **Solution :**

```
def recompose_relatif(dec: str) -> int:
    """
    Détermine la valeur de l'entier relatif représenté par dec dans la base 2
    lorsqu'on utilise le complément à 2.
    La chaîne doit être organisée de telle sorte que les valeurs de poids le
    plus élevé apparaissent en premier.

    HYPOTHÈSE : la chaîne est binaire et est codée sur un octet.
    REMARQUE : recompose_relatif utilise recompose.

    >>> recompose_relatif('10000000')
    -128
    >>> recompose_relatif('01111111')
    127
    >>> recompose_relatif('00000000')
    0
    """
    lg_octet = 8
```

```
    nbre = recompose(dec)
    if nbre >= 2**(lg_octet - 1):
        nbre = nbre - 2**lg_octet
    return nbre
```

3 Quelques opérations

10) Définition la fonction `addition` dont la spécification est :

```
def addition(c_1: str, c_2: str) -> str:
    """
    Retourne le résultat de l'addition des deux nombres dont les représentations
    binaires sont c_1 et c_2.

    HYPOTHÈSE : c_1 et c_2 sont des octets
    REMARQUE : Une chaîne vide est retournée si la somme des deux nombres ne peut
    pas être représentée avec un octet.

    >>> addition('00000000', '00000001')
    '00000001'
    >>> addition('00000001', '00000001')
    '00000010'
    >>> addition('00000001', '00000011')
    '00000100'
    >>> addition('11111111', '00000001')
    ''
    """
```

▶ **Solution :**

```
def addition(c_1: str, c_2: str) -> str:
    """
    Retourne le résultat de l'addition des deux nombres dont les représentations
    binaires sont c_1 et c_2.

    HYPOTHÈSE : c_1 et c_2 sont des octets
    REMARQUE : Une chaîne vide est retournée si la somme des deux nombres ne peut
    pas être représentée avec un octet.

    >>> addition('00000000', '00000001')
    '00000001'
    >>> addition('00000001', '00000001')
    '00000010'
    >>> addition('00000001', '00000011')
    '00000100'
    >>> addition('11111111', '00000001')
    ''
    """
    reponse = ""
```

```
    i = len(c_1) - 1
    retenue = 0  # Première retenue est forcément nulle
    while i >= 0:
        nbre_1 = int(c_1[i])
        nbre_2 = int(c_2[i])
        somme = (nbre_1 + nbre_2 + retenue) % 2
        if (nbre_1 + nbre_2 + retenue) > 1:
            retenue = 1
        else:
            retenue = 0
        reponse = str(somme) + reponse
        i = i - 1
    if retenue != 0:  # Si dernière retenue pas 0, nombre pas codable sur un octet
        reponse = ""
    return reponse
```

11) Cette définition de l'addition est-elle valable avec des entiers relatifs ?

▶ **Solution :** Oui !

```
>>> addition('01111111', '10000000')
'11111111'
```

12) Définir la fonction `decale_vers_gauche` dont la spécification est :

```
def decalage_vers_gauche(dec: str, n: int) -> str:
    """
    Décale tous les bits de n rangs vers la gauche dans dec, représentation d'un
    entier. Des 0 sont ajoutés aux places créées sur la partie droite.

    HYPOTHÈSE : dec est un octet (ce qui n'est pas forcément le cas de la réponse).

    >>> decalage_vers_gauche('00000001', 1)
    '000000010'
    >>> decalage_vers_gauche('00000011', 1)
    '000000110'
    >>> decalage_vers_gauche('00000011', 3)
    '00000011000'
    """
```

▶ **Solution :**

```
def decalage_vers_gauche(dec: str, n: int) -> str:
    """
    Décale tous les bits de n rangs vers la gauche dans dec, représentation d'un
    entier. Des 0 sont ajoutés aux places créées sur la partie droite.

    HYPOTHÈSE : dec est un octet (ce qui n'est pas forcément le cas de la réponse).

    >>> decalage_vers_gauche('00000001', 1)
```

```
    '000000010'
    >>> decalage_vers_gauche('00000011', 1)
    '000000110'
    >>> decalage_vers_gauche('00000011', 3)
    '00000011000'
    """
    for i in range(n):
        dec = dec + "0"
    return dec
```

13) Quelle opération mathématique réalise la fonction `decalage_vers_gauche` sur le nombre passé en argument ?

► **Solution :** Multiplication par 2^n.

14) Définir la fonction `decalage_vers_droite` dont la spécification est :

```
def decalage_vers_droite(dec: str, n: int) -> str:
    """
    Décale tous les bits de n rangs vers la droite dans dec, représentation d'un
    entier. Des 0 sont ajoutés aux places créées sur la partie gauche.

    HYPOTHÈSE : dec et la réponse sont des octets.
    REMARQUE : decalage_vers_droite doit utiliser octet.

    >>> decalage_vers_droite('00000011', 1)
    '00000001'
    >>> decalage_vers_droite('10000011', 2)
    '00100000'
    """
```

► **Solution :**

```
def decalage_vers_droite(dec: str, n: int) -> str:
    """
    Décale tous les bits de n rangs vers la droite dans dec, représentation d'un
    entier. Des 0 sont ajoutés aux places créées sur la partie gauche.

    HYPOTHÈSE : dec et la réponse sont des octets.
    REMARQUE : decalage_vers_droite doit utiliser octet.

    >>> decalage_vers_droite('00000011', 1)
    '00000001'
    >>> decalage_vers_droite('10000011', 2)
    '00100000'
    """
    dec = dec[:len(dec) - n]
    dec = octet(dec)
    return dec
```

15) Quelle opération mathématique réalise la fonction `decalage_vers_droite` sur le nombre passé en argument ?

► **Solution :** Division par 2^n.