Le codage en base 64

Objectifs :

- découvrir un moyen de transmission des documents binaires, l'usage de la base 64
- comprendre le principe de ce codage
- le programmer
- mettre en œuvre les dictionnaires : **rappels**

Scénario de la séance

- la veille envoyer un courrier électronique à tous les participants avec une pièce jointe (non textuelle), par exemple une **image**
- en début de séance inviter tous les participants à lire ce courrier à l'aide d'un webmail ou autre logiciel de lecture de courriers
- enregistrer le courrier dans un fichier
- lire le contenu de ce fichier correspondant uniquement à l'image avec un simple éditeur de textes : **le consulter** (cliquer le bouton `Raw`); observer que le texte est constitué de lignes de longueur identique comportant chacune 76 caractères, sauf éventuellement la dernière. L'enregistrer dans un dossier intitulé projet_codage_base64 en le nommant `image.txt`.
- s'apercevoir que la pièce-jointe est représentée sous forme textuelle, le mail ne pouvant transporter que des caractères ASCII (American Standard Code for Information Interchange), d'ailleurs on peut remarquer l'encodage des caractères accentués du message proprement dit.
- seuls 64 symboles apparaissent (les 26 lettres de l'alphabet latin non accentué en versions majuscules et minuscules, les 10 chiffres, les caractères `+` et `/`)
- utiliser dans un shell **sous Linux** la commande `base64` pour coder/décoder (en se plaçant d'abord dans le dossier de l'image grâce aux commandes `ls` et `ld`) :

```
base64 --decode texte_image.txt > image_originale.png
base64 image_originale.png > texte_image2.txt
```

Sous Windows, accéder à `Invite de commande` dans `Système Windows`, puis `dir` au lieu de `ls` et écrire `certutil -decode NSI_2022\projet_codage_base64\image.txt NSI_2022\projet_codage_base64\image.png`

- observer le principe du codage en base 64 : 3 octets, donc 24 bits, consécutifs de la donnée binaire à encoder sont découpés en 4 paquets de 6 bits, chaque paquet de 6 bits étant associé à l'un des 64 symboles (26=64).
- la question du bourrage : que faire si la taille en octets de la donnée binaire n'est pas un multiple de 3 (ou, dit autrement, si le nombre de bits de la donnée binaire n'est pas un multiple de 6) ? on complète avec un ou 2 `=`.
- nous allons programmer un codeur puis un décodeur base 64.

Description du codage en base 64

Le codage Base64 permet de transformer toute donnée binaire en une suite de symboles d'un alphabet de 64 symboles données dans la table ci-dessous.

Sextet (déc.)	Code	Sextet (déc.)	Code	Sextet (déc.)	Code	Sextet (déc.)	Code
000000 (0)	A	000001 (1)	B	000010 (2)	C	000011 (3)	D
000100 (4)	E	000101 (5)	F	000110 (6)	G	000111 (7)	H
001000 (8)	I	001001 (9)	J	001010 (10)	K	001011 (11)	L
001100 (12)	M	001101 (13)	N	001110 (14)	O	001111 (15)	P
010000 (16)	Q	010001 (17)	R	010010 (18)	S	010011 (19)	T
010100 (20)	U	010101 (21)	V	010110 (22)	W	010111 (23)	X
011000 (24)	Y	011001 (25)	Z	011010 (26)	a	011011 (27)	b
011100 (28)	c	011101 (29)	d	011110 (30)	e	011111 (31)	f
100000 (32)	g	100001 (33)	h	100010 (34)	i	100011 (35)	j
100100 (36)	k	100101 (37)	l	100110 (38)	m	100111 (39)	n
101000 (40)	o	101001 (41)	p	101010 (42)	q	101011 (43)	r
101100 (44)	s	101101 (45)	t	101110 (46)	u	101111 (47)	v
110000 (48)	w	110001 (49)	x	110010 (50)	y	110011 (51)	z
110100 (52)	0	110101 (53)	1	110110 (54)	2	110111 (55)	3
111000 (56)	4	111001 (57)	5	111010 (58)	6	111011 (59)	7
111100 (60)	8	111101 (61)	9	111110 (62)	+	111111 (63)	/

Exemple : coder trois octets en quatre symboles

La façon de procéder à ce codage est très simple : on découpe les 24 bits correspondant à ces trois octets en quatre paquets de six bits (ou sextets).

Chaque paquet de six bits correspond à un symbole ; on a en effet 2^6 sextets

possibles soit 64 sextets possibles, autant que de caractères ASCII.

Voici un exemple du codage en base 64 du triplet d'octets (18, 184, 156) :

```
  18       184      156
00010010 10111000 10011100
000100 101011 100010 011100
  E      r      i      c
```

Ainsi le triplet d'octets (18, 184, 156) est encodé par les quatre symboles `Eric`.

Coder un fichier binaire en base64 consiste donc à coder chaque bloc de trois octets consécutifs par ce procédé.

Codage de blocs incomplets

Que faire si la taille du fichier binaire n'est pas multiple de trois octets ?

Le dernier bloc peut ne contenir qu'un ou deux octets. Sans le démontrer mais pour se le prouver, on peut afficher les restes des 10 premiers multiples de 8 par la division euclidienne par 6 en réalisant la liste de ces restes par compréhension : on voit que les restes sont toujours 0, 2 ou 4.

```
>>> [(i*8)%6 for i in range(10)]
[0, 2, 4, 0, 2, 4, 0, 2, 4, 0]
```

Voyons donc les deux cas de figure.

1.**Exemple d'un bloc de deux octets :** on a 16 bits de données. On rajoute **2 bits fictifs nuls** : c'est le *bourrage* ou *remplissage* (*padding* en anglais). Cela permet d'avoir 18 bits soit 3 sextets codés par trois symboles. Pour le signifier, on ajoute un symbole particulier, le symbole `=` qui signale qu'il y a deux bits fictifs ajoutés. Voici un exemple avec le couple d'octets (18, 184) :

```
  18       184
00010010 10111000
000100 101011 1000|00
  E      r      g
```

Ainsi le couple d'octets (18, 184) est encodé par les quatre symboles `Erg=`, le dernier symbole signalant qu'un bourrage de deux bits a été effectué.

2.**Exemple d'un bloc d'un seul octet :** il manque alors deux octets, et les huit bits doivent être complétés par **4 bits fictifs nuls** pour pouvoir former deux sextets codés par deux symboles. On ajoute deux symboles `=` pour signaler la présence de quatre bits fictifs. Voici un exemple avec l'octet singleton 18 :

```
  18
00010010
000100 10|0000
  E      g
```

Ainsi l'octet 18 est encodé par les quatre symboles `Eg==`.

I] Fonction `to_base64` programmée de deux méthodes différentes

On se munit d'une table définissant les 64 symboles de la base 64.

```
BASE64_SYMBOLS = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
                  'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
                  'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
                  'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
                  'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
                  'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
                  'w', 'x', 'y', 'z', '0', '1', '2', '3',
                  '4', '5', '6', '7', '8', '9', '+', '/']
```

Armé de cette table et des opérations logiques présentées ci-dessus, il est facile de programmer l'encodage d'un tuple d'octets (donnés chacun de façon décimale) en une chaîne de symboles de la base 64, ainsi que l'opération inverse de décodage.

Code pour la vérification des docstring :

```
if __name__ == '__main__':
    import doctest
    doctest.testmod(optionflags=doctest.NORMALIZE_WHITESPACE | doctest.ELLIPSIS, verbose=Tru
```

Première méthode :
Réalisez la fonction `to_base64_slice(n_uplet)` en suivant les indications ci-dessous
a) Compléter la fonction `conversion_binaire_decimal(mot_binaire)` qui retourne la valeur décimale d'un mot binaire.

```
python def
conversion_binaire_decimal(mot_binaire):
""" Renvoie
la valeur
décimale
d'un mot
binaire
param :
mot_binaire
return :
int >>>
conversion_binaire_decimal("00101")
5 """
```

b) Compléter la fonction `conversion_decimal_binaire_6bits(nombre)` qui retourne un mot binaire écrit sur 6 bits à partir de la valeur décimale de celui-ci. Il faut envisager l'ajout de un ou plusieurs 0 pour aller jusque 6 bits.

```
python def
conversion_decimal_binaire_6bits(nombre):
""" Renvoie
le code
binaire sur
6 bits
param : dec
: int
return :
str >>>
conversion_decimal_binaire_6bits(3)
'000011'
"""
```

c) Compléter la fonction `conversion_decimal_binaire_8bits(nombre)` qui retourne un mot binaire écrit sur 8 bits à partir de la valeur décimale de celui-ci. Il faut envisager l'ajout d'un ou de plusieurs 0 pour aller jusque 8 bits.

```
python def conversion_decimal_binaire_8bits(nombre):
""" Renvoie le code binaire sur 8 bits
param : dec : int
return : str >>> conversion_decimal_binaire_8bits(3)
'00000011'
"""
```

d) Créer un dictionnaire appelé `equivalence` qui associe aux 64 symboles leur code binaire écrit sur 6 bits. On réalisera la création de ce dictionnaire **par compréhension**.

Pour gagner du temps, on utilisera ici la liste appelée `lettres` donnée ci-dessous (qui respecte la position des caractères dans la table) et on reprendra la fonction précédente `conversion_decimal_binaire_6bits(dec)`. En d'autres termes, on réalise un dictionnaire où la clé est un caractère situé à l'indice de position i dans la liste `lettres` et la valeur la conversion binaire sur 6 bits de ce même entier i.

```
python
lettres=["A","B","C","D","E","F","G","H","I","J","K","L","M","N",
```

Exemple à vérifier dans la console :

```
python >>>
equivalence['B']
'000001'
```

e) Compléter la fonction `get_in_dictionary(sixtet)` qui renvoie la clé du dictionnaire `equivalence` à partir de la donnée du sixtet.

```
python def get_in_dictionary(sixtet):
""" Renvoie la clé du dictionnaire à partir du sixtet
param : str
return : str >>> get_in_dictionary('000001')
'B' """
```

f) Compléter la fonction `sequence_binaire(n_uplet)` qui renvoie un mot binaire à partir d'un tuple constitué d'octets (on obtient ainsi un mot de 24 bits à partir d'un triplet de nombres). Pour cela, utiliser la fonction précédente : `conversion_decimal_binaire_8bits(dec)`.

```
python def
sequence_binaire(n_uplet):
""" Renvoie
un mot
binaire à
partir d'un
tuple
d'octet
param :
n_uplet :
tuple
return :
str >>>
sequence_binaire((105,86,66))
'011010010101011001000010'
"""
```

g) Réaliser un découpage dans la chaîne de caractères appelée `sequence` en utilisant le `slicing` d'une chaîne de caractères. Des explications sont données ci-après.

```
python def
to_base64_slice(n_uplet):
'''
convertit
le tuple
d'octets en
une chaîne
de symboles
:param
n_uplet:
tuple : une
séquence
d'octets
:return:
str : la
chaîne de
symboles de
la base 64
représentant
le tuple
d'octets
:CU: les
entiers du
tuple tous
compris
entre 0 et
255
:Exemples:
>>>
to_base64_slice((18,
184, 156))
'Eric' >>>
to_base64_slice((18,
184))
'Erg=' >>>
to_base64_slice((18,))
'Eg==' '''
```

Exemple de `slicing` :

```
python >>>
s="parapluie"
>>>
s[4:9]#4
compris, 9
exclu
'pluie'
```

Faire un schéma peut nous aider. Trois cas sont en effet à distinguer.
Premier cas : le nombre de bits est un multiple de 6, on va chercher les équivalents dans le dictionnaire pour chaque découpe de 6.
Deuxième cas : le reste de la division du nombre de bits par 6 est égal à 4. On ajoute '00' à la séquence, on va chercher les équivalents dans le dictionnaire pour chaque découpe de 6, et on ajoute au code '='.
Troisième cas : le reste de la division du nombre de bits par 6 est égal à 2. On ajoute '0000' à la séquence, on va chercher les équivalents dans le dictionnaire pour chaque découpe de 6, et on ajoute au code '=='.

Deuxième méthode :

On se propose de réaliser maintenant la fonction d'une autre méthode en écrivant la fonction `to_base64(n_uplet)_logique` qui utilise les opérateurs logiques présentés plus haut.

```
def to_base64_logique(n_uplet):
    '''
    convertit le tuple d'octets en une chaîne de symboles
    :param n_uplet:  tuple : une séquence d'octets
    :return: str : la chaîne de symboles de la base 64 représentant le tuple d'octets
    :CU: les entiers du tuple tous compris entre 0 et 255
    :Exemples:
    >>> to_base64_logique((18, 184, 156))
    'Eric'
    >>> to_base64_logique((18, 184))
    'Erg='
    >>> to_base64_logique((18,))
    'Eg=='
    '''
```

Rappels sur les opérations logiques sur les entiers

Python dispose d'opérateurs logiques sur les entiers : les opérations booléennes classiques sont étendues aux bits de l'écriture binaire des entiers, avec la convention que le bit `0` correspond à la valeur booléenne `False`, et le bit `1` à `True`.

1. **Et :**

   ```
   >>> 131 & 19
   3
   >>> 0b10000011 & 0b10011
   3
   ```

2. **Ou :**

   ```
   >>> 131 | 19
   147
   >>> 0b10000011 | 0b10011
   147
   ```

3. **Ou exclusif :**

   ```
   >>> 131 ^ 19
   144
   ```

```
>>> 0b10000011 ^ 0b10011
144
```

En plus de ces opérations logiques, Python propose deux opérateurs de décalage

1. **Décalage à gauche (multiplication par deux):**

```
>>> 131 << 1
262
>>> 131 << 2
524
```

2. **Décalage à droite (division par deux) :**

```
>>> 131 >> 1
65
>>> 131 >> 2
32
```

Exemple pour comprendre le principe :

Admettons que le tuple soit (105,86,66) et que la sequence_binaire obtenue soit '011010010101011001000010', c'est-à-dire, en faisant apparaître les sextets à l'aide de points insérés pour plus de lisibilité : '011010.010101.011001.000010'. Supposons que l'on souhaite extraire la deuxième découpe de 6 bits en partant de la droite soit 011001, sextet correspondant au symbole Z.

```
>>> 0b011010010101011001000010#on va travailler avec la séquence de bits dont la valeur déc
>>> 0b111111#c'est le sixtet formé de bits égaux tous à 1 dont la valeur décimale est 63
63
>>> bin(63<<6)
'0b111111000000'# cette opération permet de décaler de 6 bits vers la gauche le mot binaire
>>> bin(6903362 & (63<<6))# l'opération logique ET appliquée sur la sequence_binaire permet
'0b11001000000'# il ne reste plus maintenant qu'à éliminer les six 0 de droite
>>> bin((6903362 & (63<<6))>>6)# on décale pour cela de 6 bits vers la droite
'0b11001'# on est bien parvenu à extraire la découpe de 6 bits recherchée
>>> conversion_decimal_binaire_6bits((6903362 & (63<<6))>>6)#pour bien récupérer l'écriture
'011001'
>>> get_in_dictionary(conversion_decimal_binaire_6bits((6903362 & (63<<6))>>6))
'Z'
```

On pourra proposer une conversion de la droite vers la gauche.

II] Fonction `from_base64` programmée de deux méthodes différentes

```
def from_base64(b64_string):
    '''
    convertit une chaîne de symboles en un tuple d'octets
    :param : b64_string: (str) une chaîne de symboles de la base 64
    :return: (tuple) un tuple d'octets dont b64_string est la représentation en base 64
```

```
    :CU: les caractères de b64_string sont dans la table ou le symbole =
    :Exemple:
    >>> from_base64('Eric')
    (18, 184, 156)
    >>> from_base64('Erg=')
    (18, 184)
    >>> from_base64('Eg==')
    (18,)
    '''
```

On pourra envisager de créer une liste que l'on transformera en tuple pour renvoyer un tuple.

Exemple de conversion de liste en tuple :

```
>>> liste=[3,4,5]
>>> n_uplet=tuple(liste)
>>> n_uplet
(3, 4, 5)
```

On reproduira la méthode du slicing ou/et la méthode des opérateurs logiques.