Raphael Mottelet – Bachelor G1

TP implémentation de l'algorythme en base 64

1 Code source:

```python
BASE64_CHARS = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

def char_to_ascii(char):
    ascii_code = ord(char)
    print(f"char_to_ascii('{char}') -> {ascii_code}")
    return ascii_code

def ascii_to_bin(ascii_code):
    bin_str = f"{ascii_code:08b}"
    print(f"ascii_to_bin({ascii_code}) -> '{bin_str}'")
    return bin_str

def bin_to_base64char(bin_str):
    index = int(bin_str, 2)
    base64_char = BASE64_CHARS[index]
    print(f"bin_to_base64char('{bin_str}') -> '{base64_char}'")
    return base64_char

def base64char_to_bin(base64_char):
    index = BASE64_CHARS.index(base64_char)
    bin_str = f"{index:06b}"
```

```
    print(f"base64char_to_bin('{base64_char}') -> '{bin_str}'")
    return bin_str

def bin_to_ascii(bin_str):
    ascii_code = int(bin_str, 2)
    print(f"bin_to_ascii('{bin_str}') -> {ascii_code}")
    return ascii_code

def ascii_to_char(ascii_code):
    char = chr(ascii_code)
    print(f"ascii_to_char({ascii_code}) -> '{char}'")
    return char

def encode_base64(input_str):
    ascii_codes = [char_to_ascii(char) for char in input_str]
    bin_str = ''.join([ascii_to_bin(ascii_code) for ascii_code in ascii_codes])
    bin_chunks = [bin_str[i:i+6] for i in range(0, len(bin_str), 6)]
    bin_chunks[-1] = bin_chunks[-1].ljust(6, '0')
    base64_chars = [bin_to_base64char(chunk) for chunk in bin_chunks]
    base64_str = ''.join(base64_chars)
    while len(base64_str) % 4 != 0:
        base64_str += '='
    print(f"encode_base64('{input_str}') -> '{base64_str}'")
    return base64_str

def decode_base64(base64_str):
    padding_count = base64_str.count('=')
    base64_str = base64_str.rstrip('=')
    bin_str = ''.join([base64char_to_bin(char) for char in base64_str])
    bin_chunks = [bin_str[i:i+8] for i in range(0, len(bin_str), 8)]
    if padding_count:
        bin_chunks = bin_chunks[:-padding_count]
    ascii_codes = [bin_to_ascii(chunk) for chunk in bin_chunks]
    decoded_chars = [ascii_to_char(ascii_code) for ascii_code in ascii_codes]
    decoded_str = ''.join(decoded_chars)
    print(f"decode_base64('{base64_str}') -> '{decoded_str}'")
    return decoded_str

def main():
    input_str = input("Entrez la chaîne à encoder en Base 64: ")
    encoded_str = encode_base64(input_str)
    print(f"Chaîne encodée: {encoded_str}")
```

```
    encoded_input = input("Entrez la chaîne Base 64 à décoder: ")
    decoded_str = decode_base64(encoded_input)
    print(f"Chaîne décodée: {decoded_str}")

if __name__ == "__main__":
    main()
```

2 Exemple d'utilisation

Encodage d'une chaine de caractère en base 64:

```
PS C:\Users\Raph\Desktop\GitHub\repo's for ESIEE-IT courses\Security_Cryptography_Courses\TP2> python.exe .\base64.py
Entrez la chaîne à encoder en Base 64: salut salut
```

Retour du programme avec la chaine encodé, puis un input proposant de décoder une chaine de caractères:

```
encode_base64('salut salut') -> 'c2FsdXQgc2FsdXQ='
Chaîne encodée: c2FsdXQgc2FsdXQ=
Entrez la chaîne Base 64 à décoder:
```

En proposant au programme de décoder la chaine résultant de l'encodage de "salut salut" (c2FsdXQgc2FsdXV1dXQ=), l'on obtiens alors le message initial encodé (salut salut).

```
decode_base64('c2FsdXQgc2FsdXQ') -> 'salut salut'
Chaîne décodée: salut salut
PS C:\Users\Raph\Desktop\GitHub\repo's for ESIEE-IT courses\Security_Cryptography_Courses\TP2>
```

3 Fonctionnement

Dans cette premiere partie du code, on déclare les caractères qui seront utilisés pour l'encodage en base 64 (tous les caractères pour la base 64).

```
BASE64_CHARS = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

Bien qu'a la fin de mon programme, c'est par mon main que débute mon algo. Dans ce main, je met toute les conditions pour que mon utilisateur puisse rentrer les input voulus, par exemple au début la chaine à encoder en base 64 puis son résutat, puis inversement si mon utilisateur veut reconvertir le méssage encodé en 64 pour le décoder.

```
def main():
    input_str = input("Entrez la chaîne à encoder en Base 64: ")
    encoded_str = encode_base64(input_str)
    print(f"Chaîne encodée: {encoded_str}")

    encoded_input = input("Entrez la chaîne Base 64 à décoder: ")
    decoded_str = decode_base64(encoded_input)
    print(f"Chaîne décodée: {decoded_str}")
```

Petit détail par rapport au main, cette conditon vérifie que mon main est bien le début de code à executer. En gros ici, **__name__** est une variable intégrer à python qui me permet de vérifier le contexte d'execution d'un module, et donc ici de vérifier si ma condition est vraie (mon main est bien mon main) et que si c'est le cas alors mon programme commence par le main.

```
if __name__ == "__main__":
    main()
```

Ici j'ai déclaré une fonction qui prend en paramètre un caractère, puis retourne son code ASCII via la fonction ord, ou ma variable ascii_code prend en mémoire la valeur de ord(char). Puis, j'utilise un printf pour afficher le processus. Ma fonction retourne ensuite ma variable contenant la valeur ascii de ma chaine.

```
def char_to_ascii(char):
    ascii_code = ord(char)
    print(f"char_to_ascii('{char}') -> {ascii_code}")
    return ascii_code
```

Dans la même idée que la fonction précédente, cette fonction permet de prendre un valeur ascii pour la convertir en binaire. Ici, plus exactement en une chaine de charactere de 8 bit (correspondant à l'ascii donné, un caractere ascii étant définis sur 8 bit).

J'utilise une formated string python pour directement insérer mon ascii à convertir, via ma variable possédant en mémoire la valeur ascii de ma chaine. Un petit printF pour le déboggage, puis je retourne ma variable bin_str (donc une string), dont la valeur est cette fois en binaire.

```
def ascii_to_bin(ascii_code):
    bin_str = f"{ascii_code:08b}"
    print(f"ascii_to_bin({ascii_code}) -> '{bin_str}'")
    return bin_str
```

Dans ce code, je segmente ma chaine définis sur 8 bit en une chaine définis en 6 bit pour correspondre à la base 64 (2^6). En transformant ma valeur en integer puis en la replacant en base 10, cela me permet d'obtenir de charactere correspondant en base 64. Pour rentrer dans les détails, j'indique dans mon index que ma chaine actuelle est en binaire (argument 2) puis en le faisant passer en base 10 je le fais correspondre au caractere 64 donné.

Pareil, déboggage avec un petit printf, puis je retourne mon caractere.

```
def bin_to_base64char(bin_str):
    index = int(bin_str, 2)
    base64_char = BASE64_CHARS[index]
    print(f"bin_to_base64char('{bin_str}') -> '{base64_char}'")
    return base64_char
```

Ici même principe mais chemin inverse, on pars du caractere en base 64 pour le faire passer en binaire

```
def base64char_to_bin(base64_char):
    index = BASE64_CHARS.index(base64_char)
    bin_str = f"{index:06b}"
    print(f"base64char_to_bin('{base64_char}') -> '{bin_str}'")
    return bin_str
```

De même, on pars en suite du binnaire pour trouver l'ascii correspondant sur 8bit

```
def bin_to_ascii(bin_str):
    ascii_code = int(bin_str, 2)
    print(f"bin_to_ascii('{bin_str}') -> {ascii_code}")
    return ascii_code
```

Puis enfin, je prend mon ascii pour le faire passer sur le caractère correspondant (encore une fois, même principe que le début de mon code mais pris à l'envers)

```
def ascii_to_char(ascii_code):
    char = chr(ascii_code)
    print(f"ascii_to_char({ascii_code}) -> '{char}'")
    return char
```

Alors ici ma fonction encore une chaine de caractere en base84, puis elle convertis les dis caractere en équivalent ascii. Puis ensuite en chaine binaires de 8 bits, puis comme mon code précédent effectue une segmentation pour effectuer une conversion en 64, dans le cas de la segmentation cela se passe dans ma boucle for. Ma fonction ajoute des caracteres de remplissage juste au cas ou la chaine retourné n'est pas assez longue.

Cette fonction comporte une petite gestion d'erreur, comme dans Dans `bin_chunks[-1].ljust(6, '0')` ou par exemple, je rajoute des 0 si la longueur est inferieure à 6 bits pour pouvoir assurer la bonne conversion en base 64. J'ai aussi par exemple ma boucle while portée sur la longueur de la chaine, qui permet de s'assurer que la chaine soit un multiple de 4 (via le modulo, si tout se passe bien c'est égal à 0) et réponde aux exigences de la base 64, ou je print un caractere en cas de besoin dans cet optique (ici un =).

```
def encode_base64(input_str):
    ascii_codes = [char_to_ascii(char) for char in input_str]
    bin_str = ''.join([ascii_to_bin(ascii_code) for ascii_code in ascii_codes])
    bin_chunks = [bin_str[i:i+6] for i in range(0, len(bin_str), 6)]
    bin_chunks[-1] = bin_chunks[-1].ljust(6, '0')
    base64_chars = [bin_to_base64char(chunk) for chunk in bin_chunks]
    base64_str = ''.join(base64_chars)
    while len(base64_str) % 4 != 0:
        base64_str += '='
    print(f"encode_base64('{input_str}') -> '{base64_str}'")
    return base64_str
```

Ici même principe que la fonction encore_base mais à l'envers, ou je commence par compter les carracteres de remplissage pour ensuite les dégager. Puis je transforme mes charactere de base 64 en binaire, et ainsi de suite.

```
def decode_base64(base64_str):
    padding_count = base64_str.count('=')
    base64_str = base64_str.rstrip('=')
    bin_str = ''.join([base64char_to_bin(char) for char in base64_str])
    bin_chunks = [bin_str[i:i+8] for i in range(0, len(bin_str), 8)]
    if padding_count:
        bin_chunks = bin_chunks[:-padding_count]
    ascii_codes = [bin_to_ascii(chunk) for chunk in bin_chunks]
    decoded_chars = [ascii_to_char(ascii_code) for ascii_code in ascii_codes]
    decoded_str = ''.join(decoded_chars)
    print(f"decode_base64('{base64_str}') -> '{decoded_str}'")
    return decoded_str
```