TP1 : Convertisseurs de bases

1 Échauffement : un premier script en Python3

1. En suivant votre fichier de consignes générales pour les TP, créez un répertoire de travail pour le cours de math, et un sous-répertoire pour la séance actuelle.

2. En suivant votre fichier de consignes générales pour les TP, créez un nouveau fichier de script, dans votre répertoire de travail :

   `~/BasesMath1/TP1/premierExemple.py`

   Dans ce fichier, copiez le code suivant :

```
mot="001011"
print(mot)
print("A l'indice 0 on trouve le caractere : "+str(mot[0]))
print("A l'indice 1 on trouve le caractere : "+str(mot[1]))
print("A l'indice 2 on trouve le caractere : "+str(mot[2]))
print("A l'indice 3 on trouve le caractere : "+str(mot[3]))
print("A l'indice 4 on trouve le caractere : "+str(mot[4]))
print("A l'indice 5 on trouve le caractere : "+str(mot[5]))
```

   Sauvez le fichier, puis exécutez-le avec `python3` (voir le fichier de consignes générales pour les TP). Vérifiez que le résultat s'affiche comme prévu dans le terminal.

3. Pour un informaticien, les 6 lignes "`print`" répétées dans le code précédent sont une insulte aux Dieux de la programmation ! Afin d'éviter leur fureur, on se dépêche de les remplacer par une **boucle for** :

```
mot="001011"
print(mot)
for i in range(0,6):
    print("A l'indice "+str(i)+" on trouve le caractere : "+str(mot[i]))
```

   Effectuez cette modification dans le fichier `premierExemple.py`, puis exécutez-le à nouveau.

   Commentaires importants :

   (a) Attention, en Python il est fondamental d'**indenter** (=décaler vers la droite) la ou les lignes de code correspondant à l'intérieur de la boucle for. Si elles ne sont pas indentées, Python ne les considèrera pas comme faisant partie de la boucle.

(b) Retenez bien la syntaxe de la boucle for. Lorsqu'on écrit `range(0,6)`, cela représente les entiers de 0 **inclus** à 6 **exclus**. On peut aussi simplement écrire `range(6)`.

4. Il reste un problème avec le code précédent : si on utilise un mot de longueur différente (par exemple, `"11101"`) le code précédent ne marchera plus, car il suppose que le mot en entrée est de longueur 6.

   Pour fixer ce problème, on peut utiliser la commande `len()`, qui renvoie la longueur de la variable qu'on lui donne en entrée :

```
mot="11101"
print(mot)
N=len(mot) # on stocke la longueur de la chaine 'mot' dans la variable 'N'
print("Ce mot est de longueur "+str(N))
for i in range(N):
    print("A l'indice "+str(i))
    print("on trouve le caractere : "+str(mot[i]))
```

   Effectuez cette modification dans le fichier `premierExemple.py`, puis exécutez-le à nouveau.

5. Dernier raffinement : au-lieu de rentrer la valeur de "`mot`" en dur dans le code, on peut souhaiter que ce soit l'utilisateur lui-même qui donne le mot au moment de l'appel du programme. Pour cela, il suffit de remplacer la première ligne du code précédent par

```
mot=input("Entrer un nombre binaire : ")
```

   Effectuez cette modification, et vérifiez que vous pouvez maintenant rentrer le mot de votre choix, lors de l'appel du script.

2 Caractères vs. nombres en Python3

Créez un nouveau ficher de script (toujours dans votre répertoire de travail), dans lequel copier et exécuter les exemples de code ci-dessous :

```
a = 3
b = 4
c = "pytho"
d = "n"
e = "3"
```

Lorsque vous déclarez une variable, Python lui associe un **type**. Dans l'exemple ci-dessus, les variables a et b ont le type **int** (nombres entiers). Les variables b, c et d ont le type **str** (chaînes de caractères), à cause des guillemets utilisés pour leur définition.

Python peut avoir des comportements différents en fonction du type de la variable. Par exemple :

```
print(a+b)
print(c+d+e)
```

Au vu de l'exemple ci-dessus, répondez aux deux questions suivantes :

- Que fait l'opérateur "+" entre des variables de type `int` ?
- Que fait l'opérateur "+" entre des variables de type `str` ?

Si on essaye de mélanger des variables de types différents, on a généralement des problèmes :

```
1 print(e+b)
2 print(c+d+a)
```

On voit ainsi que les variables a et e *ne sont pas interchangeables*. La variable a contient le **nombre** 3, qu'on peut utiliser dans des calculs. La variable e contient le **caractère** `"3"`, qu'on peut utiliser dans un texte.
Lorsque c'est possible, Python peut cependant **convertir** des variables, par exemple transformer une `str` en `int`, ou réciproquement :

```
1 print(int(e)+b)
2 print(c+d+str(a))
```

En résumé :

- Concatener deux chaînes de caractères : "bon"+"jour" vaut "bonjour"
- Convertir une chaîne de caractères en entier : int("5") vaut 5
- Convertir un entier en une chaîne de caractères : str(5) vaut "5"
- Calcul de la longueur d'une chaîne de caratères : pour `s="bonjour"`, `len(s)` vaut 7.
- Accès au caractère d'indice *i* d'une chaîne de caractères : pour `s="bonjour"`, `s[3]` vaut "j" et `s[0]` vaut "b"

3 Représentations décimales, binaires et hexadécimales

Il faut bien distinguer un **nombre** de sa **représentation** dans une base donnée. Par exemple, appelons *N* le *nombre de jours entre deux pleines lunes*. Même un homme des cavernes, sans système d'écriture, pouvait concevoir ce nombre. Il lui suffit de graver une coche chaque soir, entre 2 pleines lunes successives. Voici le nombre *N* :

Un **nombre entier**, au sens mathématique "pur" du terme, c'est juste ça : un tas de bâtons.

Par contre, au moment où on souhaite *décrire* ce nombre, communiquer sa valeur à d'autres personnes, etc., les bâtons ne seront pas pratiques ! C'est pourquoi on a inventé des **systèmes de numération** permettant de décrire les nombres plus rapidement qu'avec les bâtons. Le système de numération le plus courant est bien sûr la base 10. Le nombre *N* en base 10 s'écrit :

Dans le système de numération binaire, le *même* nombre N s'écrit

$$''11011''$$

Dans le système de numération héxadécimal, le *même* nombre N s'écrit

$$''1B''$$

mais c'est *toujours le même nombre* ! Seule sa représentation change.

Informatiquement, la disctinction entre le nombre pur et sa représentation va correspondre à deux types de variables différentes :

- Le **nombre** pur correspond au type **int**. Algorithmiquement, un `int` est comme un "tas de bâtons" que Python peut manipuler dans des calculs (en ajouter, en multiplier, etc.).
- Par contre, les **représentations** du nombre sont des **str**, autrement dit : du texte. Dans ce TP, les nombres binaires et hexadécimaux seront représentés par des chaînes de caractères avec le bit de poids faible à droite. Par exemple, la variable suivante (une chaîne de caractères) :

  $$s = '' \, 1B''$$

  nous servira à représenter un nombre en hexadécimal.
  La seule exception est la représentation *décimale* : celle-ci, Python la comprend directement. Il n'y a donc pas besoin de la représenter avec une **str**, on peut juste dire *N=27* et Python comprend bien sûr de quel nombre on parle !

Le but de ce TP sera d'écrire des **convertisseurs de base**, c'est-à-dire d'écrire des programmes qui prennent un nombre *décrit dans base* (par exemple : décimal) et qui en déduisent sa représentation *dans une AUTRE base* (par exemple : binaire).

4 Exercices

Maintenant, c'est à vous ! Créez un nouveau fichier de script (par exemple, `exercices.py`) et écrivez dedans le code permettant de résoudre chaque exercice. Si vous préférez, vous pouvez même créer un fichier de script différent pour chaque exercice.

Voici quelques commandes mathématiques en Python dont vous aurez besoin :

- Calcul du reste d'une division euclidienne : `5%2` vaut 1
- Calcul du quotient d'une division euclidienne : `5//2` vaut 2
- Calcul de x à la puissance n : `2**3` vaut 8

Exercice 1 (Algorithme de Hörner). Cet algorithme prend en entrée une *représentation* (dans une base donnée), et calcule le *nombre* correspondant. Il est illustré dans les slides de cours.

1. Sur papier : appliquez l'algorithme de Hörner pour calculer le nombre dont la représentation binaire est 11100110. (Attention ! L'algorithme de Hörner n'est pas forcément la manière la plus intuitive de faire ce calcul. Mais c'est bien lui qu'il faut appliquer.)
2. Sur ordinateur : écrivez un programme Python3 qui prend en entrée une représentation en base 2 (sous forme de `str`) et renvoie le nombre correspondant, avec l'algorithme de Hörner.

   Voici une trame possible (les petits points sont à remplir par vos soins) :

```
binaire = "11100110"  # (on peut en tester d'autres)
nombre = 0
for i in ... :
    nombre = ...

print(nombre)      # on est sorti de la boucle, affiche le résultat
```

Exercice 2 (Algorithme des divisions). Cet algorithme prend un *nombre* en entrée, et calcule sa *représentation* dans une base donnée. C'est l'opération inverse de l'algorithme de Hörner. Il est illustré dans les slides de cours.

1. Sur papier : appliquez l'algorithme des divisions pour calculer la représentation binaire du nombre 89. (Attention ! L'algorithme des divisions n'est pas forcément la manière la plus intuitive de faire ce calcul. Mais c'est bien lui qu'il faut appliquer.)
2. Sur ordinateur : écrivez un programme Python3 qui prend un nombre en entrée, et renvoie une **str** contenant sa représentation en base 2, avec l'algorithme des divisions.

   Voici une trame possible (les petits points sont à remplir par vos soins) :

```
nombre = 89        # (on peut en tester d'autres)
binaire = ""       # va stocker la représentation binaire
while ... :
    bitdedroite = ...
    binaire = ... + binaire
    nombre = ...

print(binaire)     # on est sorti de la boucle, affiche le résultat
```

Exercice 3 (Lecteur de caractères hexadécimaux).

1. Écrivez une **fonction** Python qui prend en entrée un *nombre* entre 0 et 15, et qui renvoie le *caractère* hexadecimal correspondant. Voici la trame générale à remplir :

```
def ma_fonction(nombre):
    ...     # (code de la fonction, sans doute sur plusieurs lignes)
    ...
    c = ... # valeur du caractère à renvoyer
    return c
```

   Une fois écrite votre fonction, testez-la : `ma_fonction(0)`, `ma_fonction(13)`, etc.

2. Que devrait-il se passer si vous appelez, par exemple, `ma_fonction(18)` ? Comment votre fonction réagit-elle à ce genre d'erreur ?
3. En général, une fonction doit toujours avoir un *nom qui indique à quoi elle sert*. Trouvez donc un nom plus approprié que "`ma_fonction`" pour la fonction que vous venez d'écrire.
4. Écrivez une seconde fonction qui fait le travail dans l'autre sens : qui prend en entrée un *caractère* hexadecimal, et renvoie le *nombre* correspondant.

Exercice 4 (Convertisseurs hexadécimaux).

1. Écrivez un programme Python3 qui prend en entrée une représentation hexadécimale (sous forme de `str`) et renvoie le nombre correspondant, avec l'algorithme de Hörner.
2. Écrivez un programme Python3 qui prend un nombre en entrée, et renvoie une **str** contenant sa représentation hexadécimale, avec l'algorithme des divisions.

Exercice 5 (Binaire vs. hexadécimal).

Il existe un lien particulièrement simple entre les représentations *binaire* et *hexadécimale*.

1. Sur papier : trouvez la représentation hexadécimale du nombre dont la représentation binaire est "10110101". Même question pour le nombre "1000011011011111".
2. Sur papier : trouvez la représentation binaire du nombre dont la représentation hexadécimale est "GAG".
3. Sur ordinateur : vérifiez vos réponses, en utilisant les différentes fonctions de conversion des exercices précédents.

Exercice 6 (Algorithmes alternatifs).

1. L'*algorithme parallèle* prend en entrée une *représentation* (dans une base donnée), et calcule le *nombre* correspondant. Il a donc le même but que l'algorithme des Hörner vu plus haut. Écrivez un programme Python3 qui implémente l'algoritme parallèle.

2. L'*algorithme glouton* prend un *nombre* en entrée, et calcule sa *représentation* dans une base donnée. Il a donc le même but que l'algorithme des divisions vu plus haut. Écrivez un programme Python3 qui implémente l'algorithme glouton.