Algo & Prog, avec Python

(L1-Sciences)

TP n° 3, Automne 2016

Exercice 3.1 Programmez une fonction `map_string(f,s)` prenant une fonction $f : \{\text{caractères}\} \rightarrow \{\text{caractères}\}$ et une chaîne `s`. Le résultat est la nouvelle chaîne obtenue en appliquant la fonction `f` à chaque caractère de `s`. Exemple :

```
def transforme(c) :
    if c == 'b' : return 'x'
    return c
```

```
> map_string(transforme,'benabo !')
'xenaxo !'
```

Exercice 3.2 Le codage des messages secrets selon Jules César consistait à choisir une clé entière `k` dans `[1,25]` pour fabriquer à partir d'un message `msg` un nouveau message **codé** avec la technique suivante. Chaque lettre majuscule de `msg` est décalée de `k` positions vers la droite (l'alphabet est **circulaire** : après 'Z' on revient sur 'A'). Les autres caractères du message sont laissés intacts !

a) Programmez la fonction `code_cesar(msg,k)` qui retourne le message codé. En utilisant la fonction `map_string` de l'exercice 1, vous vous ramènerez à programmer la fonction `code_char(c,k)` de codage d'un seul caractère `c`. Exemple :

```
code_cesar('ENVOYEZ 36 HOMMES !',3) → 'HQYRBHC 36 KRPPHV !'
```

b) Sur le même modèle, programmez la fonction `decode_cesar(msg,k)` qui prend un message codé par a) et retourne le message en clair. Exemple :

```
decode_cesar('HQYRBHC 36 KRPPHV !',3) → 'ENVOYEZ 36 HOMMES !'
```

c) Défi urgent : décidez le message 'JLGUI XRJFZC' dont Jules a perdu la clef !

Exercice 3.3 *IMPORTANT*. Programmez une fonction `add_strbin(s1,s2)` prenant deux chaînes `s1` et `s2` contenant des chiffres binaires, et retournant la chaîne représentant le résultat binaire de l'addition de `s1` et `s2`. Vous simulerez via une boucle la technique **avec retenue** utilisée à la main, en partant de la droite. Vous prendrez garde à la possibilité que les deux chaînes ne soient pas de même longueur, en rajoutant des 0 sur la gauche de la plus courte des deux...

```
  011
+ 101
----
1000
```

```
add_strbin('11','101') → '1000'
```

Exercice 3.4 La fonction mathématique *sinus* est non seulement *indéfiniment dérivable* mais en plus elle peut se représenter sous la forme d'une somme infinie convergente (les matheux disent qu'elle est *analytique* sur $\mathbf{R}$). Pour tout `x` réel :

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + \dots = \sum_{n=0}^{+\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

a) Calcul approché d'un sinus. Sans utiliser la fonction `sin` bien entendu, programmez en Python **avec une boucle for** une fonction `sinus(x,k)` retournant la somme ci-dessus jusqu'à la puissance x^{2k+1} incluse. Utilisez l'opérateur `**` ainsi que la fonction `factorial`.

$$\text{sinus}(x,k) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots + (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

```
>>> from math import pi
>>> sin(pi/3)           # la valeur fournie par le sin prédéfini du module math
0.8660254037844386
>>> sinus(pi/3,4)      # pour x = pi/3, calcul de x - x^3/3! + x^5/5! - x^7/7! + x^9/9!
0.8660254450997811
```

b) Afin d'*accélérer le calcul*, reprogrammez la fonction `sinus(x,k)` en éliminant l'opérateur puissance `**` ainsi que la fonction `factorial` qui refait des calculs inutiles à chaque tour de boucle. *Ah, ces programmeurs obsédés par la vitesse...*

Exercices complémentaires

Exercice 3.5 Programmez une fonction `table_mult(n)` prenant un entier `n` tel que $1 \leq n \leq 9$, et affichant la table de multiplication par `n`.

Exercice 3.6 Programmez itérativement une fonction `alphabet()` dont le résultat est la chaîne :

```
'abcdefghijklmnopqrstuvwxy'
```

Exercice 3.7 *D'après le cours Python de 1ère année du MIT¹.*

La recherche de chaînes (*string matching*) est intéressante dans plusieurs disciplines, comme la biologie, dont un problème courant consiste à comprendre la structure des molécules d'ADN, et le rôle de structures spécifiques dans le fonctionnement de la molécule. Une séquence ADN est représentée par une suite $c_0c_1c_2 \dots$ de caractères choisis parmi les quatre nucléotides : adenine (A), cytosine (C), guanine (G) et thymine (T). Par exemple, la chaîne de caractères 'AAACAACCTTCGTAAGTATA' représente un brin d'ADN.

Une manière d'étudier un brin d'ADN consiste à y rechercher une séquence précise, avec l'idée qu'une structure identique induira des effets identiques. Nous allons pénétrer un tout petit peu dans cette idée. Vous utiliserez la méthode prédéfinie `find` sur les chaînes² (non vue en cours) plutôt que `index` qui provoque une erreur en cas d'échec.

```
'atgacatgcacaagtatgcat'.find('atgc') → 5
```

```
'atgacatgcacaagtatgcat'.find('foo') → -1
```

- Programmez une fonction `count_substring_match(s1,s2)` qui prend deux chaînes en argument et retourne le nombre de fois que `s2` apparaît comme sous-chaîne de `s1`.

```
count_substring_match('atgacatgcacaagtatgcat', 'atgc') → 2
```

¹ Basé sur l'excellent livre de John Guttag (sur amazon.fr, cherchez 'Guttag Python'). Attention, il est rédigé en Python 2 (ce n'est pas très grave)...

² Nous avons implémenté la *fonction* `position` dans le cours 4 page 9 comme cas particulier de `find`.

Algo & Prog, avec Python

(L1-Sciences)

TD n° 3, Automne 2016

Exercice 3.1 L'exercice suivant est un complément de cours. Cherchez la documentation sur les fonctions *bin* et *hex* dans les docs de Python 3.2, dans un livre, dans Google, où vous voudrez...

- Le *é* accentué a pour unicode 233 décimal. Comment demande-t-on au toplevel Python de voir l'écriture *binaire* (base 2) de 233 ?
- Sur papier maintenant, quelle est l'écriture binaire de l'entier qui s'écrit 55 en décimal ? Vérifiez votre réponse au toplevel.
- Sur papier, quel est le résultat de l'addition 11010 + 10111 en binaire ? Vérifiez votre réponse au toplevel.
- Quelle est l'écriture *hexadécimale* (base 16) de l'entier qui s'écrit 164 en décimal ? Vérifiez-le au toplevel.
- Sur papier, quel est le résultat de l'addition 3F + A2 en hexadécimal ? Vérifiez votre réponse au toplevel.

Exercice 3.2 En utilisant l'idée d'*épluchage* d'un entier [cours 3 page 2], programmez les fonctions suivantes :

- La fonction `somch(n)` prenant un entier $n \geq 0$ [ceci est garanti et n'est donc pas à vérifier], et retournant la somme des chiffres de n en base 10. *Ne passez pas par les chaînes de caractères !*
- La fonction `somchbin(n)` retournant cette fois la somme des chiffres de n en binaire. Exemple :
`somchbin(3456) → 4`
- La fonction `int2strbin(n)` prenant un entier $n \geq 0$ et retournant la chaîne contenant l'écriture binaire de n . Exemple :
`int2strbin(3456) → '110110000000'`
- La fonction inverse `strbin2int(s)` prenant une chaîne s contenant l'écriture binaire d'un entier $n \geq 0$, et retournant n . Exemple :
`strbin2int('110110000000') → 3456`
- La fonction `renverser(n)` prenant un entier $n \geq 0$ et retournant l'entier obtenu en prenant les chiffres de n en sens inverse. Exemple :
`renverser(34560) → 6543`

Exercice 3.3 Reprogrammez une autre version de la fonction `eplucher(n)` du cours page 2 sans utiliser les opérations `%` et `//`, mais en passant par une chaîne de caractères.

Exercice 3.4 a) Reprogrammez la fonction suivante en remplaçant la boucle `for` par une boucle `while` :

```
def foo(x,a,b) : # on suppose a < b
    for i in range(a,b) :
        x = x + i
    print(x)
```

- Reprogrammez cette même fonction sans utiliser aucune boucle !