1 Les chaînes de caractères

1.1 Introduction

Une *chaîne de caractères* est un ensemble ordonné de caractères. Les caractères peuvent être des lettres (majuscules ou minuscules), des chiffres, des signes de ponctuation, des symboles ou même des caractères dit « spéciaux » sur lesquels nous reviendrons plus tard dans ce chapitre.

En Python, on représente une chaîne de caractères en plaçant les caractères la constituant entre une paire de guillemets ou une paire d'apostrophes :

```
In [1]: ch1 = "Hello World!"

In [2]: ch2 = 'Ceci est une chaîne de caractères'
```

Les deux écritures, équivalentes, existent par commodité. En effet, pour que l'interpréteur s'y retrouve, il n'est pas possible [1] de mettre de guillemets dans une chaîne encadrée par des guillements (ni d'apostrophes dans une chaîne encadrée par des apostrophes) :

```
In [3]: ch3 = "une telle chaîne contenant " est incorrecte"

SyntaxError: invalid syntax
```

En revanche, on peut écrire :

```
In [4]: ch4 = "c'est une chaîne correcte"

In [5]: ch5 = 'ceci aussi est une "bonne" chaîne'
```

1.2 Manipulation des chaînes de caractères

Les chaînes de caractères, en tant qu'ensemble ordonnés de caractères, ont beaucoup de choses en commun avec les listes. Par exemple, on peut créer une nouvelle chaîne de caractères en concaténant deux chaînes existantes avec l'opérateur `+` :

```
In [6]: ch1 + ch2
Out[6]: 'Hello World!Ceci est une chaîne de caractères'
```

Ou bien créer une chaîne de caractères par répétition d'une autre chaîne de caractères avec l'opérateur de multiplication `*` :

```
In [7]: ch1 * 3
Out[7]: 'Hello World!Hello World!Hello World!'
```

On peut également, comme une liste, déterminer la longueur d'une chaîne de caractères avec la fonction `len` :

```
In [8]: len(ch1)
Out[8]: 12
```

On peut enfin, comme pour une liste, obtenir un caractère de la chaîne en précisant sa position entre crochets :

```
In [9]: ch1[6]
Out[9]: 'W'

In [10]: ch1[-2]
Out[10]: 'd'
```

On constate ici que, pour une chaîne également, le premier caractère se trouve à la position d'indice 0. Un indice négatif permet, comme pour une liste, de désigner un caractère en partant de la fin de la chaîne.

On peut également construire une sous-chaîne ou une chaîne regroupant certains des caractères en utilisant un « slice » :

```
In [11]: ch1[:5]
Out[11]: 'Hello'

In [12]: ch1[8:10]
Out[12]: 'rl'

In [13]: ch1[-6:]
Out[13]: 'World!'

In [14]: ch1[1::2]
Out[14]: 'el ol!'
```

Au contraire des listes, qui peuvent contenir des objets de tous types, une chaîne de caractères ne contient que des caractères ! Python n'a pas de type particulier pour représenter un caractère unique, on utilise simplement une chaîne de caractères de longueur égale à 1.

1. du moins pas directement, nous y reviendrons.

Une autre différence, majeure, entre les listes et les chaînes de caractères est qu'**une chaîne de caractères n'est pas modifiable** (on dit qu'elle est *immutable*) :

```
In [15]: ch1[6] = "w"

TypeError: 'str' object does not support item assignment
```

Ainsi, pour obtenir une chaîne de caractères sans majuscule au mot `World`, il faudra créer une *nouvelle* chaîne, en écrivant par exemple :

```
In [16]: ch1 = ch1[:6] + "w" + ch1[7:]

In [17]: ch1
Out[17]: 'Hello world!'
```

2 Égalité de chaînes

2.1 Opérateur d'égalité

Comme pour la plupart des objets Python, on peut tester si deux chaînes sont égales avec l'opérateur `==`.

Deux chaînes sont égales si et seulement si elles contiennent exactement les mêmes caractères, dans le même ordre (majuscules et minuscules ont de l'importance) [2].

```
In [18]: "Hello" == "World"
Out[18]: False

In [19]: "Hello" == "Hello"
Out[19]: True

In [20]: "Hello" == "hello"
Out[20]: False

In [21]: chaine = "Hello"

In [22]: chaine == "Hello"
Out[22]: True

In [23]: chaine == "chaine"
Out[23]: False
```

2. On pourrait également envisager d'utiliser l'opérateur `is` sur deux chaînes de caractères, mais comme une chaîne n'est pas un objet mutable, cela n'a pas d'intérêt ; c'est même fortement déconseillé car, pour des raisons d'implémentation, les résultats sont imprévisibles.

L'opérateur `!=` permet, comme pour d'autres objets, de tester si deux chaînes sont différentes.

```
In [24]: "Hello" != "World"
Out[24]: True

In [25]: chaine != "chaine"
Out[25]: True
```

Nous verrons un peu plus tard dans ce cours qu'il est également possible d'utiliser les opérateurs de comparaison `<`, `<=`, `>` et `>=` pour comparer deux chaînes de caractères.

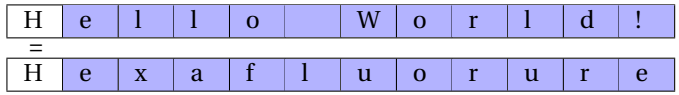
2.2 Une fonction pour tester l'égalité de chaînes

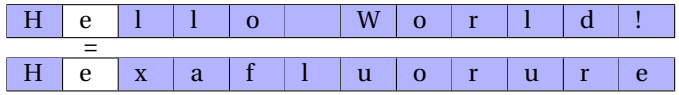
Bien que cela ne présente en pratique aucun intérêt, nous allons écrire une fonction testant l'égalité de deux chaînes, afin de mieux comprendre ce qui se passe lors d'une comparaison. On peut raisonner de cette façon :

> Si les longueurs des deux chaînes sont différentes
> alors les chaînes sont différentes
>
> Pour tout caractère de la première chaine
> si le caractère correspondant de la seconde est différent
> alors les chaînes sont différentes
>
> Si aucune différence n'a été trouvée, les chaînes sont identiques

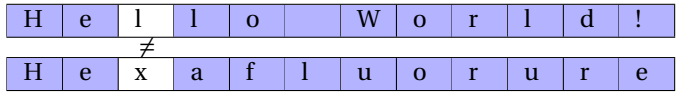
Commencer par comparer les longueurs des deux chaînes permet (outre d'identifier rapidement des chaînes différentes) de s'assurer que l'on ne fera référence qu'à des caractères à l'intérieur des deux chaînes dans la boucle.

Par exemple, si l'on compare les deux chaînes (de même longueur) `"Hello World!"` et `"Hexafluorure"`, les deux premiers caractères sont identiques :





Mais le troisième permet de conclure que les chaînes sont différentes :



La traduction de l'algorithme en langage Python est immédiate :

```
def Identiques(chaine1, chaine2) :
    """Retourne True si chaine1 est identique à chaine2, False sinon"""

    # On compare les longueurs
    if len(chaine1) != len(chaine2) :
        return False

    # Puis on compare les caractères deux à deux
    for i in range(len(chaine1)) :
        # Si deux caractères sont différents, les chaînes le sont aussi
        if chaine1[i] != chaine2[i] :
            return False

    # Si on ne trouve pas de différence, elles sont identiques
    return True
```

Comme dans le cas de la recherche d'un élément dans une liste, il faut bien prendre garde à ce que le « **return True** » se situe *en-dehors* de la boucle, comme c'est le cas dans l'algorithme proposé : on ne peut conclure que les deux chaînes sont identiques qu'après avoir constaté que *tous* les caractères étaient identiques deux à deux, et pas seulement les deux premiers !

Dans le pire des cas, pour conclure que deux chaînes sont identiques, le nombre de comparaisons qu'il nous faudra effectuer sera donc proportionnel à la longueur des chaînes.

3 Recherche d'une sous-chaîne dans une chaîne

3.1 Présentation du problème

Une autre question se pose fréquemment : est-ce qu'une chaîne de caractère `ch1` « contient » une chaîne de caractères `ch2` ? C'est-à-dire, une sous-chaîne de `ch1` est-elle égale à `ch2` ?

Comme pour les listes, on peut utiliser le mot-clé **in** pour le savoir :

```
In [26]: ch1 = "Hello World!"

In [27]: "World" in ch1
Out[27]: True

In [28]: "kangourou" in ch1
Out[28]: False
```

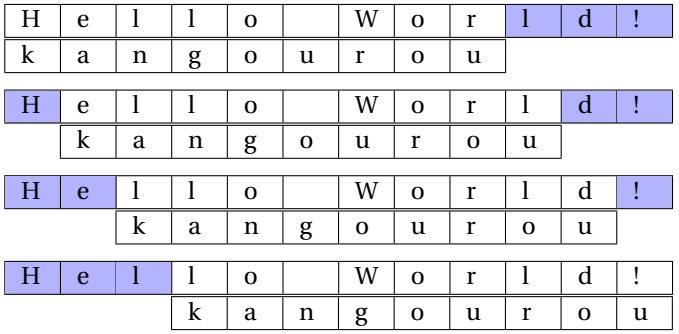
Comme pour les comparaisons, pour que le résultat soit **True**, il faut que la sous-chaîne soit parfaitement identique : une espace supplémentaire, une majuscule à la place d'une minuscule, ou toute autre modification même mineure et le résultat sera **False**.

```
In [29]: "hello" in ch1
Out[29]: False
```

3.2 Algorithme naïf

Pour mieux comprendre ce qui se passe, nous allons oublier l'existence du mot-clé « in » et écrire nous-même une fonction `Contient(ch1, ch2)` qui testera si `ch2` est une sous-chaîne de `ch1`.

La méthode la plus simple consiste à considérer toutes les sous-chaînes de `ch1` de longueur adéquate et de les comparer avec `ch2`. Par exemple, la chaîne `"kangourou"` pourrait se trouver à quatre endroits possibles dans la chaîne `"Hello World!"` :



Aussi, pour toutes les positions possibles, on compare la chaîne `"kangourou"` et la sous-chaîne correspondante dans `"Hello World!"`. Aucune de ces comparaisons ne donnant une égalité, on peut en conclure que la chaîne `"kangourou"` n'est pas présente dans la chaîne `"Hello World!"`.

En Python, nous écrirons donc :

```
def Contient(ch1, ch2) :
    # On envisage toutes les positions possibles pour la sous-chaîne
    for debut in range(0, len(ch1)-len(ch2)+1) :
        if ch2 == ch1[debut : debut+len(ch2)] :
            return True

    # La recherche n'a rien donné, le résultat est négatif
    return False
```

On prêtera une attention particulière ici aux bornes du `range` : le plus petit index où `ch2` peut débuter dans `ch1` serait bien évidemment 0, le plus grand serait `len(ch1)-len(ch2)`,

d'où le second argument de `range`.

On remarquera que si `ch2` est plus longue que `ch1`, on obtient immédiatement **`False`**, puisque l'on n'effectuera aucune itération.

Comme précédemment, on peut effectuer la comparaison explicitement :

```
def Contient(ch1, ch2) :
    # On envisage toutes les positions possibles pour la sous-chaîne
    for debut in range(0, len(ch1)-len(ch2)+1) :
        identique = True

        # On teste si les caractères de ch1 sont identiques à ceux
        # de la sous-chaîne ch1[debut:debut+len(ch2)]
        for pos in range(len(ch2)) :
            if ch1[debut+pos] != ch2[pos] :
                identique = False
                break

        # S'ils le sont, alors ch1 est incluse dans ch2
        if identique == True :
            return True

    # La recherche n'a rien donné, le résultat est négatif
    return False
```

On peut aussi vouloir éviter le **`break`** en utilisant un **`while`** :

```
def Contient(ch1, ch2) :
    for debut in range(0, len(ch1)-len(ch2)+1) :
        identique = True

        # On remplace ici la boucle for par une boucle while
        # afin de sortir de la boucle dès que identique != True
        pos = 0
        while identique == True and pos < len(ch2) :
            if ch1[debut+pos] != ch2[pos] :
                identique = False
            pos = pos + 1

        # On teste si l'on est sorti prématurément de la boucle ou non
        if identique == True :
            return True

    return False
```

3.3 Au-delà de l'algorithme naïf

Dans le cas de l'algorithme présenté au paragraphe précédent, si l'on ne parvient pas à trouver la chaîne `ch1` à l'intérieur de la chaîne `ch2`, l'entier `debut` prendra successivement toutes les valeurs de `0` à `len(ch2)-len(ch1)`. Puis, pour chaque valeur de `debut`, on effectue jusqu'à `len(ch1)` comparaisons.

Le nombre de comparaisons de caractères à effectuer pourrait donc, dans le pire des cas, atteindre $\big($`len(ch2)-len(ch1)+1`$\big)\times$`len(ch1)`. En pratique, il est très inférieur, car toutes les comparaisons de sous-chaînes de `ch2` à `ch1` ne peuvent pas échouer sur le dernier caractère. Toutefois, dans les cas les plus défavorables, le nombre de comparaisons peut être de l'ordre du produit des longueurs des deux sous-chaînes.

Il existe de nombreuses méthodes nettement plus efficaces (en ce sens qu'elles effectuent moins de comparaisons, donc nécessitent moins de temps pour donner le résultat) pour comparer deux chaînes de caractères, dont la connaissance n'est toutefois pas au programme. Les algorithmes plus efficaces[3] effectuent un nombre de comparaison de l'ordre de la somme des longueurs des deux chaînes plutôt que de leur produit. Elles sont en revanche plus complexes à écrire, et nécessitent des préparations avant la recherche proprement dite. L'idée générale est que, lorsqu'une comparaison échoue, on peut souvent augmenter `debut` de plus qu'une unité.

3.4 Décompte et localisation

Au-delà du **`in`** qui fournit un booléen indiquant si une chaîne est présente dans une autre, on dispose d'autres fonctions pour savoir non seulement si elles sont présentes, mais combien elles sont, et où elles se trouvent précisément.

De la même façon que pour les listes on avait la fonction `list.count(L, elem)` qui indiquait le nombre d'occurrence de `elem` dans la liste `L`, il existe une fonction `str.count(ch1, ch2)` qui indique le nombre (maximal) d'occurrences *sans recouvrement* de `ch2` dans `ch1` :

```
In [30]: s = "ABABABAB"

In [31]: str.count(s, "A")
Out[31]: 4

In [32]: s.count("AB")
Out[32]: 4

In [33]: s.count("ABA")
Out[33]: 2
```

3. Parmi les plus connus, nommés selon leurs auteurs, figurent les algorithmes de Knuth-Morris-Pratt, de Boyer-Moore et de Rabin-Karp.

On remarquera que, comme pour une liste, on peut se dispenser du `str` en tête du nom de la fonction (qui précise le type d'objet sur lequel elle agit, ici une chaîne de caractères), et le remplacer par le premier argument.

De même, on avait une fonction `list.index(L, elem)` qui indiquait la position de `elem` dans la liste `L`, et renvoyait une erreur si `elem` n'était pas présent, on a une fonction `str.index(ch1, ch2)` (qui renvoie l'index indiquant où *commence* la sous-chaîne de `ch1` égale à `ch2`) :

```
In [34]: str.index(s, "BA")
Out[34]: 1

In [35]: s.index("BA")
Out[35]: 1

In [36]: s.index("BC")
ValueError: substring not found
```

Parfois, une erreur si la chaîne n'est pas trouvée n'est pas pratique, aussi a-t-on une variante, `str.find(ch1, ch2)` qui renvoie la valeur `-1` si `ch2` n'est pas une sous-chaîne de `ch1`.

```
In [37]: s.find("BA")
Out[37]: 1

In [38]: s.find("BC")
Out[38]: -1
```

Notre propre fonction `Contient` peut être très simplement modifiée pour proposer le même comportement, il suffirait de renvoyer `debut` au lieu de **`True`**, lorsque l'on a trouvé la chaîne, et `-1` au lieu de **`False`** si ce n'est pas le cas.

Normalement, la recherche se fait de gauche à droite dans la chaîne, et retourne donc l'occurrence placée la plus à gauche. Mais les fonctions `str.rindex` et `str.rfind` permettent de faire ces recherches en partant de la fin de la chaîne de caractères.

```
In [39]: s.rfind("BA")
Out[39]: 5

In [40]: s.rindex("BA")
Out[40]: 5
```

Pour obtenir le même comportement de notre fonction `Contient`, il suffirait simplement d'inverser le `range` de la boucle **`for`** [4], ou de modifier le **`while`** pour qu'il décrémente `debut` jusqu'à atteindre 0 plutôt que de l'incrémenter à chaque étape.

4. En prenant garde aux bornes qui ne seront pas simplement échangées !

4 Entrées et sorties

4.1 Affichage, convertion d'un objet en chaîne

Nous avons déjà vu que la fonction `print` permettait d'afficher non seulement une chaîne de caractères, mais également d'autres objets. En réalité, elle ne fait qu'afficher des chaînes de caractères, mais Python est capable de transformer la plupart des objets en une chaîne de caractères les représentant.

La fonction `str` permet, au besoin, d'effectuer explicitement une telle conversion :

```
In [41]: str(42)
Out[41]: '42'

In [42]: str([ 1, 2 ])
Out[42]: '[1, 2]'

In [43]: str(abs)
Out[43]: '<built-in function abs>'
```

Parfois, on veut plus de liberté dans la fabrication d'une chaîne à partir d'éléments qui n'en sont pas. Il existe en Python plusieurs mécanismes assez puissants (et subséquemment parfois complexes) pour ce faire. L'un d'entre eux est basé sur la fonction de mise en forme `str.format(chaineFormat, ...)`. Un exemple sera beaucoup plus parlant :

```
In [44]: str.format("x = {} mètres", 12.51)
Out[44]: 'x = 12.51 mètres'
```

On constate, sur cet exemple, que la fonction a pris la chaîne-format et a remplacé le « jeton » `{}` par l'argument suivant parmi les arguments de la fonction. On peut tout à fait mettre plusieurs jetons :

```
In [45]: str.format("{} fois {} égale {}", 2, 3, 2*3)
Out[45]: '2 fois 3 égale 6'
```

La puissance de ce système réside dans le fait que l'on peut contrôler la façon dont vont apparaître les arguments à la place des accolades en glissant des informations à l'intérieur de celles-ci. Par exemple, on peut spécifier le nombre de chiffres à afficher (chiffres significatifs) :

```
In [46]: str.format("pi vaut environ {:.3}", math.pi)
Out[46]: 'Pi vaut environ 3.14'

In [47]: str.format("pi/100 vaut environ {:.6}", math.pi/100)
Out[47]: 'Pi vaut environ 0.0314159'
```

Il n'est pas question, ici, d'entrer dans le détail de toutes les possibilités que ce mécanisme offre, vous pourrez allez chercher les détails au besoin dans la documentation Python.

Une dernière fonction peut enfin être utile : la fonction `str.join(sep, liste)`. Elle prend en argument une chaîne `sep` et une liste de chaînes `liste` et renvoie la concaténation de tous les éléments de `liste`, en glissant la chaîne `sep` entre chacun. C'est beaucoup plus efficace que d'utiliser l'opérateur `+` s'il y a plus de deux chaînes à concaténer (car dans le cas de l'opérateur `+`, on créerait beaucoup de chaînes successives, une pour chaque concaténation).

Ainsi, on peut écrire :

```
In [48]: str.join(", ", [ "A", "B", "cde" ])
Out[48]: 'A, B, cde'

In [49]: ", ".join([ "A", "B", "cde" ])
Out[49]: 'A, B, cde'
```

Et donc pour concaténer une liste de chaînes (sans rien insérer entre chaque chaîne), on peut utiliser :

```
In [50]: str.join("", [ "A", "B", "cde" ])
Out[50]: 'ABcde'

In [51]: "".join([ "A", "B", "cde" ])
Out[51]: 'ABcde'
```

La dernière écriture n'est pas la plus claire, mais elle est très fréquente dans les programmes en Python, et vous la croiserez sans doute fréquemment.

Toutes ces fonctions permettent donc de mettre en forme des résultats, qui seront ensuite affichés pour le bénéfice de l'utilisateur du programme, ou bien encore enregistrés dans un fichier. Il ne faut pas oublier que si un ordinateur a généralement pour tâche principale d'effectuer des calculs, s'il ne communique pas avec son environnement, cela ne présentera guère d'intérêt.

4.2 Entrées, convertion d'une chaîne en objet

Pour la même raison, un ordinateur a besoin de pouvoir recevoir des informations de l'extérieur, que ce soit de la part de l'utilisateur, d'un fichier, du réseau, etc.

La fonction permettant de demander à l'utilisateur d'entrer des données est la fonction `input(invite)` qui prend un seul argument, une chaîne de caractères `invite` qui sera affichée pour le bénéfice de l'utilisateur. Puis le programme attend que l'utilisateur entre quelque chose au clavier, et le valide avec la touche « entrée ». Ce que l'utilisateur aura entré est alors retourné *sous la forme d'une chaîne de caractères.*

```
In [52]: res = input("Veuillez entrer un nombre :")
Veuillez entrer un nombre :
```

(ici, l'utilisateur entre par exemple 42 au clavier, et l'invite de commande n'apparaît à nouveau que lorsqu'il a pressé « entrée » et validé la ligne)

```
In [53]: res
Out[53]: '42'
```

Le résultat, encore une fois, est bien une chaîne de caractères, quoi que l'utilisateur ait souhaité entrer[5] ! Puisque le résultat est toujours une chaîne de caractères, on aura souvent besoin de le traduire en un autre type. Par exemple, la fonction `int` transforme son argument en entier, et la fonction `float` en flottant.

```
In [54]: int("42")
Out[54]: 42

In [55]: float("42")
Out[55]: 42.0

In [56]: int("3.14")
Out[56]: 3
```

Il existe une fonction `list` qui transforme son argument en liste, mais...

```
In [57]: list("[1, 2]")
Out[57]: ['[', '1', ',', ' ', '2', ']']
```

... si elle retourne bien une liste, ce n'est peut-être pas la liste que l'on espérait ! La fonction construit une liste, contenant les différents caractères constituant la chaîne de caractères.

En fait, il existe une fonction « automatique », qui s'efforce de deviner le type d'objet que désigne la chaîne, et effectue la conversion correspondante, c'est la fonction `eval` :

```
In [58]: eval("42")
Out[58]: 42

In [59]: eval("3.14")
Out[59]: 3.14

In [60]: eval("[1, 2]")
Out[60]: [1, 2]
```

5. Attention, cependant, `input` ne fonctionne pas exactement de la même façon en Python 2. L'équivalent de `input` en Python 2 est la fonction `raw_input`, tandis que `input` en Python 2 revient à appeler la fonction `input` de Python 3k puis à lui appliquer la fonction `eval`, et ne retourne donc pas toujours une chaîne de caractères.

Toutefois, si l'on connaît le type que l'on souhaite obtenir, il est préférable d'utiliser une fonction convertissant explicitement la chaîne dans le type souhaité plutôt que de se fier à la fonction `eval`.

Citons enfin deux fonctions bien pratiques pour travailler avec des chaînes de caractères. La première, `str.strip`, permet de retirer les caractères « non imprimables » (espaces, tabulations, retours à la ligne, etc.) à chaque extrémité de la chaîne passée en argument :

```
In [61]: str.strip("  Blop    ")
Out[61]: 'Blop'

In [62]: chaine = "  Blop    "

In [63]: chaine.strip()
Out[63]: 'Blop'
```

La seconde est la fonction `str.split` qui « découpe » une chaîne passée en argument et retourne une liste de chaînes. Par défaut, le découpage se fait sur les espacements (qui seront supprimés), mais on peut spécifier via un argument supplémentaire un caractère qui servira de séparateur :

```
In [64]: str.split("  A, bc, DeF ")
Out[64]: ['A,', 'bc,', 'DeF']

In [64]: "  A, bc, DeF ".str.split()
Out[64]: ['A,', 'bc,', 'DeF']

In [65]: str.split("  A, bc, DeF ", ",")
Out[65]: ['  A', ' bc', ' DeF ']
```

Ces fonctions sont utiles si l'on demande à l'utilisateur de rentrer d'un seul coup plusieurs informations (par exemple des coordonnées, séparées par des virgules). Elles seront très précieuses également lorsque l'on lira un fichier, un peu plus tard, afin d'en extraire des données. On pourra par exemple écrire :

```
In [66]: chaine = " 12.34, 10.23   "

In [67]: x, y = [ float(s) for s in chaine.split(",") ]

In [68]: print("x = {} et y = {}".format(x, y))
x = 12.34 et y = 10.23
```

5 Représentation des caractères

5.1 De l'ASCII à l'Unicode

Comme les entiers et les réels, les caractères doivent être représentés en mémoire en binaire, par des suites de 0 et de 1. Très rapidement, les informaticiens se sont dit qu'il serait utile de s'entendre sur un encodage, de sorte que, pour tout le monde, la valeur binaire « 100000 » désigne une espace, et « 1000001 » la lettre A majuscule, par exemple, de sorte qu'il soit plus facile de s'échanger des textes par voie numérique.

Au début des années 1960 [6], après de nombreuses discussions, l'*American Standard Code for Information Exchange*, ou ASCII, est introduit.

Le code ASCII associe un code de 7 bits aux caractères les plus courants : notamment les majuscules de A à Z (entre 1000001 et 1011010, dans l'ordre), les minuscules correspondantes (entre 1100001 et 1111010), les chiffres (de 110000 à 111001), ainsi que divers signes de ponctuation, symboles ou opérateurs mathématiques courants.

Les codes 0000000 à 0011111 et le code 1111111 sont des codes spéciaux, de contrôle (retour à la ligne, fin de transmission, bip, tabulation, retour arrière, etc.) facilitant notamment les transmission de machine à machine.

	000...	001...	010...	011...	100...	101...	110...	111...
...0000	NULL	DLE	espace	0	@	P	‘	p
...0001	SOH	DC1	!	1	A	Q	a	q
...0010	STX	DC2	"	2	B	R	b	r
...0011	ETX	DC3	#	3	C	S	c	s
...0100	EOT	DC4	$	4	D	T	d	t
...0101	ENQ	NAK	%	5	E	U	e	u
...0110	ACK	SYN	&	6	F	V	f	v
...0111	BEL	ETB	’	7	G	W	g	w
...1000	BS	CAN	(	8	H	X	h	x
...1001	TAB	EM	)	9	I	Y	i	y
...1010	LF	SUB	*	:	J	Z	j	z
...1011	VT	ESC	+	;	K	[	k	{
...1100	FF	FS	,	<	L	\	l	|
...1101	CR	GS	-	=	M	]	l	}
...1110	SO	RS	.	>	N	^	n	~
...1111	SI	US	/	?	O	_	o	DEL

Comme on le voit, le code ASCII ne permet donc d'encoder que 95 caractères (de 0100000 à 1111110), ce qui est insuffisant pour la quasi-totalité des langues en dehors de l'anglais, aucun accent n'étant par exemple présent dans le code.

De très nombreuses extensions ont rapidement vu le jour, en général compatibles avec le code ASCII : puisque l'on travaille généralement avec des octets, on a fréquemment étendu la norme à des codes sur 8 bits, utilisant les codes 10000000 à 11111111 pour encoder des caractères supplémentaires.

6. Quoique des ajustements aient été apportés jusqu'en 1986

Seulement, 128 caractères supplémentaires, cela reste très insuffisant, et très rapidement ont été proposés des codages sur 8 bits. Seulement, il existe quantité d'extensions différentes, pratiquement autant qu'il y a de pays, et parfois davantage (d'autant que lorsque les européens se sont mis d'accord pour développer une première extension, les représentants français ont oublié de demander le œ...) Sans compter certaines langues (par exemple les langues utilisant des idéogrammes comme le chinois ou le japonais) employant bien plus que quelques centaines de caractères !

Aujourd'hui, il existe un standard universel, appelé Unicode, qui associe aux différents caractères un code numérique unique. Actuellement, 1114112 codes ont été réservés pour cet usage, et 245000 ont été attribués (couvrant ainsi plus d'une centaine de langages et d'écritures). Les codes du standard ASCII original ont été conservés en Unicode.

Le hic, c'est qu'il faudrait 21 bits pour représenter un caractère Unicode si l'on utilisait directement le code fourni par la norme. Un texte en anglais occuperait en l'état deux fois et demi à trois fois plus d'espace en mémoire ou sur un support de stockage. Il existe donc différentes variantes d'encodage pour « compresser » ces codes, connues par exemple sous le nom de UTF-8 ou UTF-16.

Nous n'entrerons pas davantage dans ces questions d'encodage. Mais il est utile de savoir que, lorsque l'on a un texte sous forme numérique, il est nécessaire de savoir quel encodage a été utilisé pour le décoder correctement et obtenir par exemples les bons accents plutôt que des caractères étranges.

5.2 Caractères et points Unicode en Python

Python dispose de deux fonctions permettant d'obtenir le code numérique (on parle « point de code Unicode » d'un caractère dans la norme Unicode, et, inversement, d'obtenir un caractère à base de son code numérique. Plutôt que de travailler avec des codes binaires, on utilise l'entier (non signé) qui lui est associé.

La fonction `ord` permet donc de transformer un caractère [7] en un entier représentant sont point de code Unicode :

```
In [69]: ord('A')
Out[69]: 65

In [70]: ord('a')
Out[70]: 97

In [71]: ord('é')
Out[71]: 233

In [72]: ord("ô")
Out[72]: 244
```

7. En fait, une chaîne de caractères contenant un unique caractère.

Pour retrouver le code binaire, on peut toujours utiliser la fonction `bin` :

```
In [73]: bin(ord('A'))
Out[73]: '0b1000001'
```

La fonction réalisant l'opération de conversion inverse, transformant un entier en caractère, est la fonction `chr` :

```
In [74]: chr(90)
Out[74]: 'Z'

In [76]: chr(234)
Out[76]: 'ê'
```

5.3 Conséquences sur les comparaisons de caractères

Le fait que l'on puisse associer de façon unique chaque caractère à un nombre fournit un ordre total sur l'ensemble des caractères : lorsque l'on compare des caractères, on compare en fait les points Unicode associés. Ainsi, on a

```
In [79]: 'a' < 'ê'
Out[79]: True
```

car 97 est bien strictement inférieur à 234.

Pour des caractères minuscules, c'est compatible avec l'ordre alphabétique usuel.

```
In [80]: 'a' < 'b'
Out[80]: True
```

Il en est de même pour les majuscules, ainsi que pour les caractères correspondant aux chiffres :

```
In [81]: 'Z' <= 'A'
Out[81]: False

In [82]: '2' <= '7'
Out[82]: True
```

Attention toutefois, une minuscule (entre a et z) est toujours supérieure à une majuscule (entre A et Z), quelle qu'elle soit :

```
In [83]: 'a' < 'B'
Out[83]: False
```

5.4 Comparaison de chaînes

Lorsque l'on compare deux chaînes de caractères, on utilise l'ordre lexicographique. C'est-à-dire que l'on commence par comparer les deux premiers caractères de chacune des deux chaînes, puis en cas d'égalité on s'intéresse au second, et ainsi de suite. Le classement est donc le même que celui d'un dictionnaire.

```
In [84]: "grand" < "petit"
Out[84]: True

In [85]: "kopeck" < "koala"
Out[85]: False
```

Si, lors de la comparaison, on arrive à l'extrémité d'une des deux chaînes sans avoir trouvé de caractères différents, alors la plus longue est la plus grande (là encore, comme dans un dictionnaire).

```
In [86]: "orange" < "oranger"
Out[86]: True
```

Attention, car cet ordre lexicographique fait que des chaînes de caractères contenant des nombres ne sont pas nécessairement ordonnées selon la valeur des nombres. Par exemple, dans l'exemple suivant, puisque `"1"` est inférieur à `"2"`, on a donc

```
In [87]: "20" < "100"
Out[87]: False
```

Le fait qu'il soit aisé de comparer des chaînes permet à beaucoup d'algorithmes de travailler avec des chaînes de caractères. Par exemple, l'algorithme de recherche dichotomique que l'on a écrit précédemment permet, sans aucune modification, de tester la présence d'une chaîne de caractères dans une liste de chaînes de caractères triées par ordre (lexicographique) croissant !

Notons au passage que, puisqu'il est possible de comparer deux chaînes de caractères avec les opérateurs de comparaison usuels, il est tout à fait possible de trier une liste de chaînes de caractères `L` par ordre lexicographique croissant en utilisant `L.sort()`.

Exercices

Ex. 1 – Sapin

Proposer une fonction prenant un entier positif non nul `h` indiquant une hauteur et affichant, dans la sortie standard, un « sapin », constitué d'un motif de forme triangulaire constitué du caractère `^` et d'un tronc constitué de deux `H`. Par exemple, pour `h=4`, on souhaite obtenir le motif suivant :

```
   ^
   ^
  ^^^
  ^^^
 ^^^^^
 ^^^^^
^^^^^^^
^^^^^^^
   H
   H
```

Ex. 2 – Pierre, papier, ciseaux

Deux amis jouent une partie de pierre-feuille-ciseaux en n manches. On rappelle que dans ce jeux, à chaque manche, les deux joueurs choisissent l'un des trois objets. S'ils choisissent le même, la manche est nulle. Sinon, on convient que la pierre bat les ciseaux, les ciseaux battent la feuille, et la feuille bat la pierre.

Chacun des deux joueurs fournit la liste de ses n coups sous la forme d'une chaîne contenant n caractères parmi `"P"`, `"F"` et `"C"` (désignant chacun l'un des trois objets du jeux). Par exemple, `"PFFCPFC"` correspond aux coups de l'un des joueurs pour une partie en $n = 7$ manches, le premier coup étant « pierre », le dernier « ciseaux »

Proposer une fonction `Match(j1, j2)` prenant en argument deux chaînes de caractères de même longueur contenant uniquement les caractères `"P"`, `"F"` et `"C"` et retournant un couple de réels `(n1, n2)` représentant le nombre de manches gagnées par chacun des joueurs.

Les plus ambitieux pourront chercher les règles du pierre-feuille-ciseaux-lézard-Spock et adapter la fonction à cette variante !

Ex. 3 – Vers lettrisés

Un *vers lettrisé*, aussi appelé *tautogramme*, est une phrase dans laquelle tous les mots débutent par la même lettre, tels que « Veni, vidi, vici. », ou bien encore « Mazarin, ministre malade, méditait même moribond malicieusement mille maltôtes. »

Proposer une fonction `EstTautogramme(chaine)` prenant en argument une chaîne de caractères contenant une phrase, et retournant un booléen indiquant si la phrase est un tautogramme (on pourra, pour simplifier, supposer que la phrase est intégralement en minuscules).

Note : Dans la pratique, il est difficile d'écrire de purs tautogrammes, et on se permet généralement quelques exceptions.

Ex. 4 – Numérotation Shadok

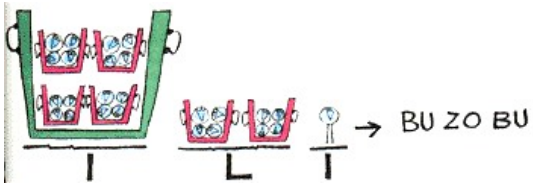
La langue Shadok contient uniquement quatre mots : « `Ga` », « `Bu` », « `Zo` » et « `Meu` ». Pour compter le nombre de Shadoks dans une réunion Shadok, le professeur Shadok a proposé la méthode de numérotation suivante :

- quand il n'y a pas de Shadoks, on dit `Ga` ;
- quand il y a un Shadok de plus, on dit `Bu` ;
- quand il y a encore un Shadok de plus, on dit `Zo` ;
- et quand il y a encore un autre, on dit `Meu`.

Si l'on met un Shadok en plus, évidement, il n'y a plus assez de mots pour les compter... Alors c'est très simple: on les jette dans une poubelle, et on dit que l'on a `Bu` poubelle. Et pour ne pas confondre avec le `Bu` du début, on dit qu'il n'y a pas de Shadok à coté de la poubelle et on écrit B`Bu Ga`.

Un Shadok de plus se retrouvera à côté de la poubelle : `Bu Bu`. Puis `Bu Zo`, `Bu Meu`. On a ensuite `Zo` poubelles et pas de Shadok à côté, donc `Zo Ga`.

On parvient rapidement à `Meu` poubelles et `Meu` Shadoks à coté. Arrivé là si l'on met un Shadok en plus, il faut une autre poubelle. Mais comme l'on n'a plus de mots pour compter les poubelles, on s'en débarrasse en les jetant dans une grande poubelle, et on écrit `Bu` grande poubelle avec pas de petite poubelle et pas de Shadok à coté : `Bu Ga Ga`. Et on continue, `Bu Ga Bu`, `Bu Ga Zo`...



`https://www.youtube.com/watch?v=lP9PaDs2xgQ`

Lorsque l'on arrive à `Meu Meu Meu` et qu'un Shadok se joint au groupe, on a trop de grandes poubelles pour pouvoir les compter, eh bien, on les met dans une super poubelle, on écrit `Bu Ga Ga Ga`, et on continue.

Écrire une fonction `NumerotationShadok(n)` qui prend en argument un entier positif `n` et retourne une chaîne de caractères correspondant à l'entier énoncé en numérotation Shadok. Par exemple :

```
In []: NumerotationShadok(75)
Out[]: "Bu Ga Zo Meu"
```