6

Représentation des textes

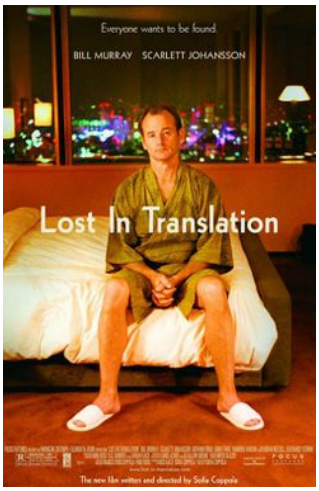
Extrait du programme

Thème : Types et valeurs de base

Contenus : Représentation d'un texte en machine.
Exemples des encodages ASCII, ISO-8859-1, Unicode

Capacités attendus : Identifier l'intérêt des différents systèmes d'encodage.
Convertir un fichier texte dans différents formats d'encodage.

Commentaires : Aucune connaissance précise des normes d'encodage n'est exigible.



Mais pourquoi ai-je reçu le message suivant ? :

```
Ceci est un texte accentuÃ© enregistrÃ© en utf8 avec des Ã© des Ã¨ des Ã§ des Ã  et mÃªme des Ã¹
```

I. Codage ASCII

a) Code ASCII

Dans les années 1950, il existait un nombre important d'encodages de caractères dans les ordinateurs, les imprimantes ou les lecteurs de carte. Tous ces encodages étaient incompatibles les uns avec les autres ce qui rendait les échanges particulièrement difficiles car il fallait utiliser des programmes pour convertir les caractères d'un encodage dans un autre.
Au début des années 1960, l'*ANSI* (American National Standards Institue) propose une norme de codage de caractères appelée ***ASCII*** (American Standard Code for Information Interchange). Cette norme définit un jeu de 128 caractères. En effet, à l'époque les ordinateurs fonctionnaient en 8 bits, et, 1 bit de parité étant conservé pour la détection des erreurs de transmission, soit $2^7 = 128$ caractères, ce qui était très largement suffisant pour coder les lettres majuscules, minuscules, chiffres et ponctuations.
Le document ci-dessous date de 1972 (*source Wikipédia*)

USASCII code chart

Bits $b_7 b_6 b_5$ →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Pour lire le codage de chaque caractère, on commence par récupérer le code binaire de la colonne correspondante, auquel on concatène [1] le code binaire de la ligne correspondante.
Ainsi, pour le caractère `A`, on est sur la colonne `100` et sur la ligne `0001`. Le code binaire ASCII du `A` est donc `100 0001`. Soit en décimale : $1 \times 2^6 + 0 \times 2^5 + ... + 0 \times 2^1 + 1 \times 2^0 = 65$.
On peut faire de même pour le codage en hexadécimal du caractère `A`. On est sur la colonne `4` et sur la ligne `1`. Le codage hexadécimal du caractère `A` est donc `41`, soit en décimale : $4 \times 16^1 + 1 \times 16^0 = 65$.

Exercice 1
Donner le codage binaire, hexadécimal puis décimal des caractères suivants :

1. **a** :
   - binaire .....
   - hexadécimal .....
   - décimal .....

1. Si, si, ce mot existe.

2. **9** :
   - binaire .....
   - hexadécimal .....
   - décimal .....

Dans la suite, nous ne donnerons que les codages en hexadécimal, ce qui est beaucoup plus court et pratique.
On remarque que la table ***ASCII*** contient plusieurs catégories de caractères :

- les chiffres de 0 à 9 (entre `30` et `39`)
- les lettres de l'alphabet latin en majuscules (entre `41` et `5A`)
- les lettres de l'alphabet latin en minuscules (entre ..... et .....)
- des signes de ponctuations (comme la virgule qui vaut `2C`, le crochet [qui vaut .....)
- des signes opérateurs arithmétiques (comme le + qui vaut `2B`, le = qui vaut .....)
- des caractères spéciaux (entre `00` et `20`)

b) Carcatères spéciaux

Voici quelques détails sur certains caractères spéciaux :

Caractère	Code hexa	Signification
HT	09	Tabulation horizontale
LF	0A	Nouvelle ligne
CR	0D	Retour chariot
FF	0C	Nouvelle page
SP	20	Espace
BS	...	Suppression
DEL	7F	Effacement

Exercice 2
Décoder le message suivant écrit en code ASCII suivants donnés en hexadécimal :
4F 75 69 20 43 48 45 46 21 (0D) 0A 42 69 65 6E 20 43 48 45 46 21
.....
.....

c) Python

En Python, on peut obtenir directement le code ***ASCII*** en décimal d'un caractère avec la fonction `ord()` et le caractère correspondant à un code avec la fonction `chr()` :

```
>>> ord('A')
65
>>> bin(ord('A')) #code en binaire
'0b1000001'
>>> hex(ord('A')) #code en hexadécimal
'0x41'
>>> chr(65)
'A'
>>> chr(0b1000001) #en binaire
'A'
>>> chr(0x41) #en hexadécimal
'A'
```

On peut aussi écrire directement en hexadécimal en rajoutant \x devant le code en hexadécimal. On appelle cette technique la technique du caractère échappée car le caractère \ s'appelle aussi le caractère d'échappement.

```
>>> print('\x4F\x75\x69\x20\x43\x48\x45\x46\x21\x0A\x42\x69\x65\x6E\x20\x43\x48\x45
    \x46\x21')
Oui CHEF!
Bien CHEF!
```

Il existe quelques raccourcis pour les caractères spéciaux. Le tableau ci-dessous en propose quelques-un :

Raccourcis	Caractère	Signification
\t	HT	Tabulation horizontale
\n	LF	Nouvelle ligne
\r	CR	Retour chariot
\f	FF	Nouvelle page
\b	BS	Suppression

Ainsi, on a :

```
>>> print('Biem\bn!\npetit\tscarabée')
Bien!
petit   scarabée
```

II. Les normes ISO 8859

Les caractères imprimables de la table ***ASCII*** se sont vites avérés insuffisants pour transmettre des textes dans les autres langues que l'anglais. En effet, rien qu'en considérant les langues reposant sur un alphabet latin, il manque dans la table ***ASCII*** de nombreux caractères comme les lettres accentuées (é, è, à, ù, ...), les symboles de monnaies (€, ...).
Pour remédier à ce problème, l'***ISO*** (Organisation de Normalisation) a proposé la norme ***ISO 8859***, une extension de l'***ASCII*** qui utilisent les huit bits de chaque octet pour représenter les caractères. Au total ce sont donc 256 caractères qui peuvent être encodés. Malgré tout, cela reste insuffisant pour représenter tous les caractères utilisés rien que dans les langues latines...
Pour représenter le plus de caractères possibles, la norme ***ISO 8859*** définit plusieurs tables de correspondances notées ***ISO 8859-***n, où n est le numéro de la table [2]. Ces tables sont toutes compatibles entre elles : les 128 premiers caractères sont ceux de la norme ***ASCII*** ; les 128 suivants sont ceux spécifiques à la table n. Les caractères identiques ont tous le même code.

Code ISO	Nom	Zone
8859-1	latin-1	Europe occidental
8859-2	latin-2	Europe centrale ou de l'est
8859-3	latin-3	Europe du sud
8859-4	latin-4	Europe du nord
8859-5		Cyrillique
8859-6		Arabe
8859-7		Grec
8859-8		Hébreu
8859-9	latin-5	Turc, Kurde
8859-10	latin-6	Révision du latin-4
8859-11		Thaï
8859-12		Devanagari [3] (projet abandonné)
8859-13	latin-7	Balte
8859-14	latin-8	Celtique
8859-15	latin-9	Révision du latin-1 (avec €)
8859-16	latin-10	Europe du sud-est

2. Il y en a 16 en tout, dont 10 uniquement pour les langues latines.
3. Écriture utilisée pour le sanskrit, le prâkrit, le hindi, le népalais, le marathi et plusieurs autres langues indiennes.

III. Unicode

Bien que les pages ***ISO-8859-***n permettent l'encodage d'un très grand nombre de caractères, elles ne conviennent pas par exemple quand on souhaite écrire un texte avec un mélange de caractères présents dans différentes pages ***ISO-8859-***n. Pour remplacer l'utilisation des pages de code, l'***ISO*** a défini un jeu universel de caractères ***UCS*** (Universal Character Set) sous la norme ***ISO-10646***. Cette norme associe à chaque caractère (lettre, nombre, idéogramme, etc...) un nom et nombre unique. Il y a aujourd'hui plus de 110 000 caractères recensés dans cette norme qui est conçu pour contenir tous les caractères de n'importe quelle langue ; la capacité maximale de la norme a été fixé à 4 294 967 295 caractères, c'est-à-dire le plus grand entier non signé représentable avec un mot de 32 bits. Par soucis de comptabilité, les 256 premiers points du code sont ceux de la norme ***ISO-8859-1*** (latin-1).
Avec un tel nombre, un encodage naïf de la norme ***ISO-10646*** utiliserait 4 octets pour représenter chaque caractère. Cependant, dans la grande majorité des cas, pour des échanges basés sur l'ancienne page latin-1, cela représenterait un énorme gâchis puisque 3 octets serait à chaque fois inutilisés en ne contenant simplement que des 0. De plus, les points de code les plus utilisés dans le monde sont rassemblés entre 0 et 65535, donc sur 2 octets.
La norme ***Unicode*** a été développée par une organisation privée à but non lucratif du même nom ***Unicode***. Elle définit plusieurs techniques d'encodage pour représenter les points de code de manière plus ou moins économique selon la technique choisie. Ces encodages sont appelés ***formats de transformation universelle*** (Universal Transformation Format) portent les noms ***UTF-***n où n indique le nombre minimal de bits pour représenter un point de code.

a) UTF-8

C'est le format le plus utilisé sous Linux, dans les protocoles réseaux et les sites Web. Comme son nom l'indique, il faut seulement 8 bits pour coder les *premiers* caractères. Il est entièrement compatible avec le standard ***ASCII***.
Actuellement, la bonne pratique est d'utiliser ***UTF-8*** dès que c'est possible.
Aussi il est vivement conseillé de toujours démarre votre code Python par cette ligne :

```
1 # -*- coding: utf-8 -*-
```

Le principe d'encodage est un peu technique (et hors programme), mais on peut le résumer ainsi :

- si le premier bit[4] est un `0`, alors il s'agit d'un caractère ***ASCII*** codé sur les 7 bits ;
- sinon, les premiers bits indiquent le nombre d'octets utilisés pour l'encodage du caractère qui peut aller de 1 à 4.

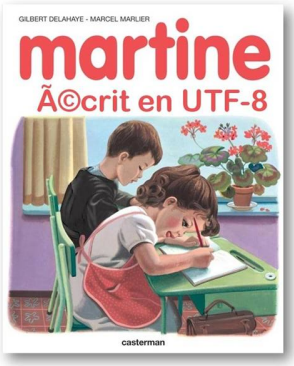
Voyons cela en Python avec le caractère `é` :

```
1 >>> 'é'.encode('utf8')
2 b'\xc3\xa9'
3 >>> bin(0xc3)
4 '0b11000011'
5 >>> bin(0xa9)
6 '0b10101001'
```

Le caractère `é` est donc codé sur 2 octets, le premier `C3`, soit `11000011` en binaire, et le deuxième `A9` soit `10101001` en binaire. Soit un codage binaire complet `11000011 10101001`.
Le bit de poids fort[5], `1`1000011 10101001, n'est pas `0`, donc il ne s'agit pas d'un code ***ASCII***.
Il y a 2 bits de poids fort non nuls, `11`000011 10101001, donc le caractère est codé sur 2 bits.

On remarque aussi, que si on ne sait pas que le fichier est encodé en ***UTF-8*** on pourra croire qu'il l'est en `ISO-8859` et donc croire qu'il n'y a pas 1 mais 2 caractères : `C3` puis `A9` soit `Ã` puis ©.
Ce qui explique quelques erreurs de *dÃ©codage* que l'on rencontre parfois[6].

4. On appelle cela le bit de poids fort.
5. Le bit le plus à gauche.
6. Oh ! Non ! Il a osÃ© !



b) UTF-16

Ce format utilise au minimum 16 bits pour représenter un caractère. Ce format permet de coder les caractères modernes les plus utilisés dans le monde.

```
>>> 'é'.encode('utf16')
b'\xff\xfe\xe9\x00'
```

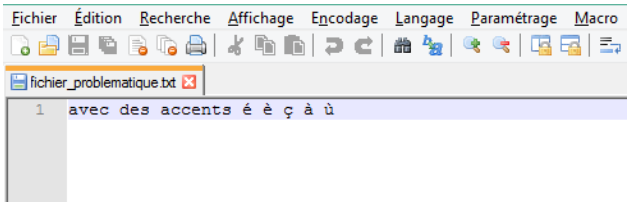
c) UTF-32

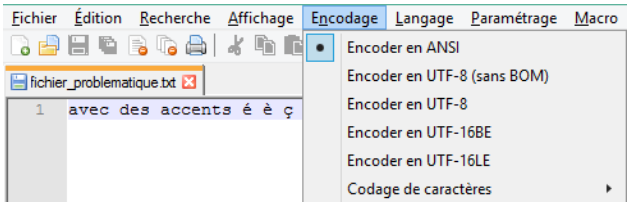
Ce format utilise au minimum 32 bits pour représenter un caractère.

```
>>> 'é'.encode('utf32')
b'\xff\xfe\x00\x00\xe9\x00\x00\x00'
```

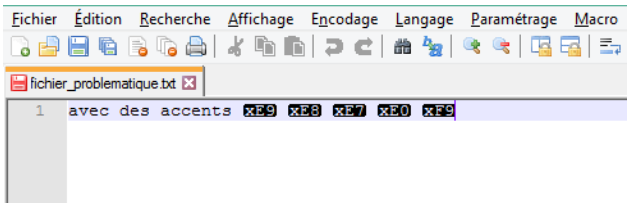
d) Erreur dans la lecture de fichiers

Ouvrir le fichier `fichier_problematique.txt` avec le logiciel **Notepad++**.





On peut connaître l'encodage utilisé avec l'outil **Encodage**. Dans notre cas, le fichier est encodé en ***ANSI***...
On peut modifier l'encodage, en sélectionnant par exemple `Encoder en UTF-8`.



Exercice 3

1. Ouvrir le fichier `fichier_problematique2.txt`.
2. Quel est l'encodage utilisé ? .....
3. Encoder en ***ANSI***. Que remarquez vous ?

.....

e) Table UNICODE

On peut trouver sur les fiches wikipédia suivantes

- Les tables de caractères
  `https://fr.wikipedia.org/wiki/Table_des_caract%C3%A8res_Unicode_(0000-0FFF)`
- des cartes, des jetons, des émoticônes
  `https://fr.wikipedia.org/wiki/Table_des_caract%C3%A8res_Unicode_(1F000-1FFFF)`

Ainsi, vous pouvez obtenir des caractères sympathiques :

```
>>> chr(0x2167) #8 en chiffre romain
>>> chr(0x04C1) #un caractère cyrillique
>>> chr(0x1F3C8) #un ballon de football américain
```

On peut aussi comme précédemment récupérer leur code ***UTF8*** :

```
>>> chr(0x2167).encode('utf8') #pour le chiffre romain
b'\xe2\x85\xa7'
>>> chr(0x04C1).encode('utf8') #pour le caractère cyrillique
b'\xd3\x81'
>>> chr(0x1F3C8) #un ballon de football américain
b'\xf0\x9f\x8f\x88'
```

IV. Exercices

Exercice 4 (*)

Donner le codage ***ASCII*** des deux chaînes de caractères au format hexadécimal :

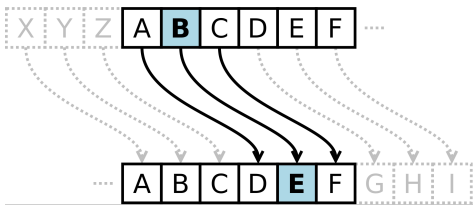
- « Bonjour le monde ! »
  .....
- « Vive
  la vie »
  .....

Exercice 5 (*)

Écrire une fonction `print_ASCII(chaine)` qui affiche les codes ***ASCII*** au format hexadécimal d'une chaîne de caractère `chaine`.
Utiliser cette fonction pour vérifier les réponses de l'exercice précédent.

Exercice 6 (***)

Le chiffre de César est historiquement la première méthode de chiffrement qui aurait été inventé par César [7].



Le texte chiffré s'obtient en remplaçant chaque lettre du texte clair original par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. Pour les dernières lettres (dans le cas d'un décalage à droite), on reprend au début. Par exemple avec un décalage de 3 vers la droite, A est remplacé par D, B devient E, et ainsi jusqu'à W qui devient Z, puis X devient A etc. Il s'agit d'une permutation circulaire de l'alphabet. La longueur du décalage, 3 dans l'exemple évoqué, constitue la clé du chiffrement qu'il suffit de transmettre au destinataire, s'il sait déjà qu'il s'agit d'un chiffrement de César.

7. Himself ! Voir l'excellent article Wikipédia sur le sujet : `https://fr.wikipedia.org/wiki/Chiffrement_par_décalage`

Écrire une fonction `cesar(chaine,clef)` prenant en argument `chaine` une chaîne de caractères uniquement entre `a` et `z` (donc pas de majuscule, pas de chiffre, aucun caractère spéciaux, pas d'espace) et la clé `clef`, un nombre entier entre 1 et 25.
Chaque caractère de la chaîne est remplacé par celui obtenu après un décalage de c caractères.
Ainsi `cesar("bon",2)` doit renvoyer `"dqp"`.
De même `cesar("xyz",2)` doit renvoyer `"zab"`.

Exercice 7 (*)
Écrire une fonction `occurrence(chaine,car)` qui prend en argument `chaine` une chaîne de caractères et `car` un caractère. Elle devra retourner le nombre de caractère égaux à `car` dans `chaine`.
Par exemple `occurrence("un premier essai",'e')` devra retourner `3`.

Exercice 8 (**)
La distance de Hamming entre deux chaînes de caractères est une notion utilisée dans de nombreux domaines (télécommunications, traitement du signal, ...). Elle est définie comme le nombre d'indices où les deux chaînes ont un caractère différent à condition que les deux chaînes soit de la même longueur.
Écrire une fonction `hamming(chaine1, chaine2)` qui prend en argument deux chaînes de caractères `chaine1` `chaine2` et retournant la distance de Hamming entre les deux chaînes. Si les deux chaînes sont de longueurs différentes, renvoyer `-1`.
Ainsi `hamming("abcd","abcr")` retourne `1` , `hamming("pomme","poire")` retourne `2` et `hamming("serviette","torchon")` retourne `-1`.

Exercice 9 (***)

1. Écrire une fonction `alenvers_mot(chaine)` prenant en argument `chaine` une chaîne de caractères constituée d'un seul mot et retournant la chaîne de caractères constituée de ce mot écrit à l'envers.
   Par exemple `alenvers_mot("prépas")` retourne `"sapérp"`.
2. Écrire une fonction `decoupe_phrase(chaine)` prenant en argument `chaine` une chaîne de caractères constituée de plusieurs mots (séparés par des espaces) et retournant la liste des différents mots.
   Par exemple `decoupe_phrase("j'ai piqué cet exercice aux prépas")` retourne la liste `["j'ai", "piqué", "cet", "exercice", "aux", "prépas"]`.
3. Écrire une fonction `alenvers_phrase(chaine)` prenant en argument `chaine` une chaîne de caractères constituée de plusieurs mots et retournant la chaîne de caractères constituée de la phrase avec les mots dans le même ordre, mais les mots écrits à l'envers.
   Par exemple `alenvers_phrase("et j'ai même pas honte")` retourne la chaîne caractères `"te ia'j emêm sap etnoh"`.

V. Caractère, chaîne de caractères en Python

a) Codage ISO 8859

La fonction `ord()` permet d'obtenir le code décimal correspondant à un caractère. La fonction `chr()` fait le contraire : elle donne le caractère correspondant au code.

```
>>> ord('é')
233
>>> bin(ord('é')) #code en binaire
'0b11101001'
>>> hex(ord('é')) #code en hexadécimal
'0xe9'
>>> chr(231)
'ç'
>>> chr(0b11100111) #en binaire
'ç'
>>> chr(0xe7) #en hexadécimal
'ç'
```

b) minuscule, MAJUSCULE

On peut convertir une chaîne en caractères minuscules avec la méthode `.lower()` et inversement en caractères MAJUSCULES avec la méthode `.upper()`.

```
>>> "Voici un sacré texte! N'est-ce pas ?".lower()
"voici un sacré texte! n'est-ce pas ?"
>>> "Voici un sacré texte! N'est-ce pas ?".upper()
"VOICI UN SACRÉ TEXTE! N'EST-CE PAS ?"
>>> "Maçon".lower()
'maçon'
>>> "Maçon".upper()
'MAÇON'
```

c) Avoir accès à chaque caractère d'une chaîne

La fonction `len()` nous donne la longueur d'une chaîne.

```
>>> len("Voici un sacré texte! N'est-ce pas ?")
36
```

On peut avoir accès à chaque caractère en comptant à partir de 0 (comme on en a maintenant l'habitude).
Pour mémoire, si on donne un nombre négatif, on compte à partir -1 pour le dernier caractère de la chaîne.

```
>>> chaine = "Voici un sacré texte! N'est-ce pas ?"
>>> chaine[0] #le premier caractère
'V'
>>> chaine[5] #le sixième caractère
' '
>>> chaine[-1] #le dernier caractère
'?'
>>> chaine[-3] #l'avant-avant dernier caractère
's'
```

Enfin, si on veut travailler sur l'ensemble des caractères de la chaîne, on peut y avoir accès avec une boucle **POUR**, autrement dit une boucle `for`.
On peut le faire avec un compteur `i` ce qui nous permet de connaître la position du caractère :

```
>>> for i in range(len(chaine)):
...     if chaine[i]=='!':
...         print("Il y a un point d'exclamation à la position numéro",i+1)
Il y a un point d'exclamation à la position numéro 21
```

On peut faire de même en accédant directement à chaque caractère de la chaîne :

```
>>> for caractere in chaine:
...     if caractere=='?':
...         print("Il y a aussi un point d'interrogation")
Il y a aussi un point d'interrogation
```

d) Vérifier si un caractère appartient à une chaîne

On pourra utiliser `in`.

```
>>> 'a' in "bonjour"
False
>>> 'a' in "au revoir"
True
```

On peut même vérifier si une chaîne de caractères est contenue dans une autre.

```
>>> 'oi' in "bonjour"
False
>>> 'oi' in "au revoir"
True
```

e) Gestion des fichiers textes avec Python

Ouverture et fermeture d'un fichier

La fonction `open()` prend deux paramètres, le nom du fichier et le mode d'ouverture :

- `'w'` pour le mode *écriture* (write). Si le fichier n'existe pas il est automatiquement créé ;
- `'r'` pour le mode *lecture* (read) ;
- `'a'` pour le mode *ajout* (append). Les écritures sont rajoutés à la fin du fichier.

Il est essentiel de fermer un fichier qui a été ouvert avec la méthode `close()`.

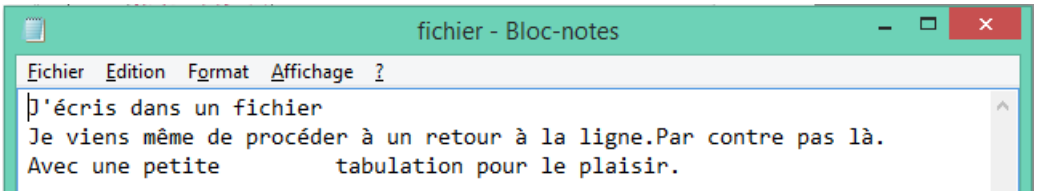
Écriture

La méthode `write()` permet d'écrire dans un fichier en prenant en paramètre une chaîne de caractères (de type `str`).

```
# -*- coding: utf-8 -*-

#création d'un fichier de type txt
f = open("fichier.txt", 'w')

#écriture dans le fichier
f.write("J'écris dans un fichier\nJe viens même de procéder à un retour à la ligne.
    ")
f.write("Par contre pas là.")
f.write("\n")
f.write("Avec une petite \ttabulation pour le plaisir.")

#fermeture du fichier
f.close()
```

L'extension `.txt` précise bien avec quel logiciel le fichier peut être ouvert, ici un éditeur de texte.
Par défaut, le fichier `fichier.txt` est créé dans le même dossier que le fichier Python.

Nous obtenons bien le résultat attendu en ouvrant le fichier avec un éditeur de texte :



Lecture

La lecture d'un fichier se fait avec la méthode `read()` :

- `f.read(n)` lit les n premiers caractères ;
- `f.read()` lit tout le fichier.

```
#ouverture d'un fichier de type txt
f = open("fichier.txt", 'r')

#lecture du fichier
lecture = f.read(10)
print(lecture) #affiche 'J'écris da'

lecture = f.read()
print(lecture) #affiche le reste du fichier à partir de 'ns un fichier ...'

#fermeture du fichier
f.close()
```

La lecture d'un fichier se fait aussi ligne par ligne avec les deux autres méthodes suivantes :

- `f.readline()` lit la ligne courante et passe à la suivante ;
- `f.readlines()` lit toutes les lignes et les renvoie dans une liste de chaînes de caractères sans modifications (on garde les `\n`, les `\t`, ...)

```
#ouverture d'un fichier de type txt
f = open("fichier.txt", 'r')

#lecture du fichier
lecture = f.readline()
print(lecture) #affiche 'J'écris dans un fichier' puis un retour à la ligne

lecture = f.readlines()
print(lecture) #affiche une liste de chaîne de caractères avec les \n et les \n
#['Je viens même de procéder à un retour à la ligne.Par contre pas là.\n', 'Avec
    une petite \ttabulation pour le plaisir.']

#fermeture du fichier
f.close()
```

VI. Exercices

Les exercices suivant utiliseront le fichier `liste_francais_source_freelang.txt` qui contient presque tous les mots utilisés en français.

Exercice 10 (*)
Quel est l'encodage utilisé ? .....

Exercice 11 (**)

1. Écrire un programme qui compte le nombre de mots présent dans cette liste. Vous devriez trouver 22 740 mots.
2. (a) La lettre `e` est la lettre la plus répandue dans la langue française. Calculer le nombre de `e` présent dans cette liste.
   (b) Donner une estimation de la fréquence d'apparition de la lettre `e` dans la langue française à partir de cette liste.
   (c) Faire la même chose pour toutes les autres lettres (non accentués) de l'alphabet puis retourner la réponse dans un fichier texte `rapport.txt`.

Exercice 12 (*)
Écrire un programme `bon_orthographe()` prenant en argument un mot sous la forme d'une chaîne de caractères et qui renvoie `True` si le mot a été écrit sans faire de faute d'orthographe ; `False` sinon.
On respectera la règle de la première lettre en majuscule pour le premier caractère des noms propres.
Par exemple `bon_orthographe("zombie")` renvoie `True` mais `bon_orthographe("Zombie")` renvoie `False`.
Autre exemple `bon_orthographe("Abidjan")` renvoie `True` mais `bon_orthographe("abidjan")` renvoie `False`.

Exercice 13 (**)
Écrire un programme `contient()` prenant en argument une chaîne de caractères sans espace et qui renvoie tous les mots contenant cette chaîne de caractères sous la forme d'un tuple.
S'il n'y a aucun mot, le programme retourne `-1`.
Par exemple `contient("nul")` renvoie `("annulation", "annulé", "annulée", "annulées", "annuler", "granulé", "granuleux", "nul", "nulle", "nulle part", "nullement", "nullité", "nuls")`.
Autre exemple `contient("xyz")` renvoie `-1`.

Exercice 14 (**)
Écrire un programme `commence_par()` prenant en argument une chaîne de caractères sans espace et qui renvoie tous les mots contenant cette chaîne de caractères commençant par cette chaîne de caractères sous la forme d'un tuple.
S'il n'y a aucun mot, le programme retourne `-1`.
Par exemple `commence_par("abb")` renvoie `("abbatial", "abbaye", "abbé", "abbesse")`.
Autre exemple `contient("xt")` renvoie `-1`.