Thème 1 : Types de bases

16 TD : Codage des caractères

Tout pour comprendre et Ã©viter les erreurs d'encodage

1. Au commencement était l'ASCII

pour *American Standard Code for Information Interchange*, créé en 1960 aux États-Unis.

Dans les années 50, il existait un nombre important d'encodages de caractères dans un ordinateur, 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 à un autre.
Pour tenter de mettre un peu d'ordre dans tout ça, en 1960, l'American Standards Association (ASA, aujourd'hui ANSI) décide de mettre un peu d'ordre dans ce bazar en créant la norme ASCII (American Standard Code for Information Interchange).

À chaque caractère est associé un nombre binaire sur 8 bits (1 octet).

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				

En faite, **seuls 7 bits sont utilisés** pour coder un caractère, le 8e bit n'est pas utilisé pour le codage des caractères. Avec 7 bits il est possible de coder jusqu'à 128 caractères ce qui est largement suffisant pour un texte écrit en langue anglaise (pas d'accents et autres lettres particulières).

- Les 32 premiers codes, de 0 à 31, ne sont pas des caractères imprimables mais des caractères "de contrôle". Par exemple le code 13 représente un retour à la ligne, et le code 7 fait produire un bip à certains ordinateurs, ce qui s'avérait utile sur les premiers IBM PC pour signaler une erreur, par exemple.
- À partir du code 32, suivent des signes de ponctuation et quelques symboles mathématiques comme ! ou + ou /, puis les chiffres arabes de 0 à 9, ainsi que les 26 lettres de l'alphabet latin, en capitales puis en minuscules.

Exercice

Décoder l'expression suivante, écrite en ASCII :

```
1101100 1100101 1110011 100000 1001110 1010011 1001001 100000
1100011 100111 1100101 1110011 1110100 100000 1101100 1100101
1110011 100000 1101101 1100101 1101001 1101100 1101100 1100101
1110101 1110010 1110011
```

Vérification avec un script Python :

Aide :

- la fonction `split(" ")` permet de décomposer une chaine de caractères en une liste, en se servant de l'espace `" "` comme caractère séparateur.
- `int("1101100",2)` permet de récupérer facilement la valeur en base 10 du nombre binaire `1101100`.
- la fonction `chr` renvoie le caractère correspondant à un entier.

```
>>> chr(78)
N
```

- La fonction `ord` de Python renvoie le code ASCII correspondant à un caractère. L'entier renvoyé est en base 10 (que l'on peut convertir enhexadécimal avec la fonction hex).

```
>>> ord('a')
97
>>> hex(ord('a))
'0x61'
```

```
msg = "1101100 1100101 1110011 100000 1001110 1010011 1001001 100000 1100011
100111 1100101 1110011 1110100 100000 1101100 1100101 1110011 100000 1101101
1100101 1101001 1101100 1101100 1100101 1110101 1110010 1110011"
msg = msg.split(' ')
s = ""
for k in msg :
    s += chr(int(k,2))
print(s)
```

2. Et le reste du monde ?

Lorsque d'autres personnes que des americains ou des anglais ont voulu s'échanger des données faisant intervenir du texte, certains caractères (é, è, à, ñ, Ø, Ö, β, 漢...) étaient manquants. Les 127 caractères de l'ASCII étaient largement insuffisants.
Il a donc été décidé de passer à... 256 caractères ! Il suffisait pour cela de coder les caractères non plus sur 7 bits mais sur 8 bits.

Ainsi naquît, a norme **ISO-8859-1**, une extension de l'ASCII qui utilise les huit bits de chaque octet pour représenter les caractères.
Cette norme va être principalement utilisée dans les pays européens puisqu'elle permet d'encoder les caractères utilisés dans les principales langues européennes (la norme ISO-8859-1 est aussi appelée "latin1" car elle permet d'encoder les caractères de l'alphabet dit "latin").
Pour ajouter à la complexité, la norme ISO-8859 définit pas moins de 15 versions différentes, pour satisfaire à tous les besoins mondiaux.

Donc après de nombreuses modifications successives (la dernière en date rajoutant par exemple le symbole €), la célèbre table **ISO 8859-15**, dite aussi **Latin-9** :

ISO/CEI 8859-15															
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*non utilisé*															
1x																
2x		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*non utilisé*															
9x																
Ax		¡	¢	£	€	¥	Š	§	š	©	ª	«	¬		®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

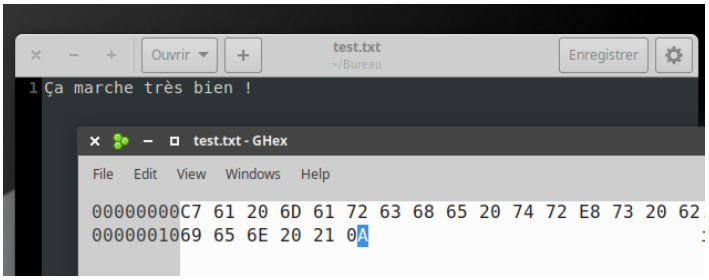
2.1 Utilisation :

Les codes sont donnés en hexadécimal :

- le caractère € correspond au code hexadécimal A4, donc au nombre décimal 164.
- le caractère A correspond au code hexadécimal 41, donc au nombre décimal 65.

65... comme en ASCII ! Oui, la (seule) bonne idée aura été d'inclure les caractères ASCII avec leur même code, ce qui rendait cette nouvelle norme rétro-compatible.

Exemple :
A l''aide de notepad écrire un texte (Ça marche très bien !). Enregistrer le avec l'encodage Latin-9.
Ce fichier est ensuite ouvert avec un éditeur hexadécimal, qui permet d'observer la valeur des octets qui composent le fichier. (Comme le fichier est un .txt, le fichier ne contient que les données et rien d'autre.)



Parfait, mais comment font les Grecs pour écrire leur alphabet ?
Pas de problème, il leur suffit d'utiliser... une autre table, appelée ISO-8859-7 :

ISO/CEI 8859-7:2003															
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*Inutilisé*															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*Inutilisé*															
9x																
Ax	NBSP	‘	’	£	€	₯	¦	§	¨	©	ͺ	«	¬	SHY		—
Bx	°	±	²	³	΄	΅	Ά	·	Έ	Ή	Ί	»	Ό	½	Ύ	Ώ
Cx	ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
Dx	Π	Ρ		Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
Ex	ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
Fx	π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	

On retrouve les caractères universels hérités de l'ASCII, puis des caractères spécifiques à la langue grecque... oui mais les Thaïlandais alors ?
Pas de problème, ils ont la ISO-8859-11 :

ISO/IEC 8859-11:2001																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*Inutilisé*															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*Inutilisé*															
9x																
Ax	NBSP	ก	ข	ฃ	ค	ฅ	ฆ	ง	จ	ฉ	ช	ซ	ฌ	ญ	ฎ	ฏ
Bx	ฐ	ฑ	ฒ	ณ	ด	ต	ถ	ท	ธ	น	บ	ป	ผ	ฝ	พ	ฟ
Cx	ภ	ม	ย	ร	ฤ	ล	ฦ	ว	ศ	ษ	ส	ห	ฬ	อ	ฮ	ฯ
Dx	ะ	ั	า	ำ	ิ	ี	ึ	ื	ุ	ู	ฺ					฿
Ex	เ	แ	โ	ใ	ไ	ๅ	ๆ	็	่	้	๊	๋	์	ํ	๎	๏
Fx	๐	๑	๒	๓	๔	๕	๖	๗	๘	๙	๚	๛				

Évidemment, quand tous ces gens veulent discuter entre eux, les problèmes d'encodage surviennent immédiatement : certains caractères sont remplacés par d'autres.

2.2 Que fait un logiciel à l'ouverture d'un fichier texte ?

Il essaie de deviner l'encodage utilisé... Parfois cela marche, parfois non.

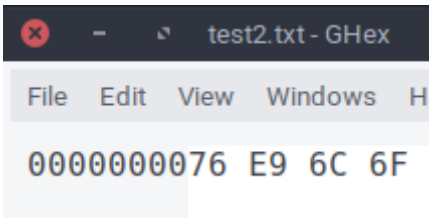
Normalement, pour un navigateur, une page web correctement codée doit contenir dans une balise `meta` le `charset` utilisé.

```
<!DOCTYPE html>
<html dir="ltr" lang="en">
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width,initial-scale=1,maximu
<link rel="preconnect" href="//abs.twimg.com" />
<link rel="preconnect" href="//api.twitter.com" />
<link rel="preconnect" href="//pbs.twimg.com" />
<link rel="preconnect" href="//t.co" />
<link rel="preconnect" href="//video.twimg.com" />
<link rel="dns-prefetch" href="//abs.twimg.com" />
<link rel="dns-prefetch" href="//api.twitter.com" />
```

Mais parfois, il n'y a pas d'autre choix pour le logiciel d'essayer de deviner l'encodage qui semble être utilisé.

Exercice :

Le mot représenté par les octets ci-dessous est-il encodé en ASCII ou en Latin-9 ?



3. Enfin une normalisation : l'arrivée de l'UTF

En 1996, le Consortium Unicode décide de normaliser tout cela et de créer un système unique qui contiendra l'intégralité des caractères dont les êtres humains ont besoin pour communiquer entre eux.

> Unicode provides a unique number for every character,
> no matter what the platform,
> no matter what the program,
> no matter what the language.

Ils créent l'**Universal character set Transformation Format : l'UTF**.
Ou plutôt ils en créent... plusieurs :

- **l'UTF-8** : les caractères sont codés sur 1, 2, 3 ou 4 octets.
- l'UTF-16 : les caractères sont codés sur 2 ou 4 octets.
- l'UTF-32 : les caractères sont codés sur 4 octets.

Pourquoi est-ce encore si compliqué ? En UTF-32, 32 bits sont disponibles, soit $2^{32} = 4294967296$ caractères différents encodables.

C'est largement suffisant, mais c'est surtout très très lourd !
D'autres encodages plus légers, mais plus complexes, sont donc proposés :

➡ Arrêtons-nous sur l'UTF-8 :

Définition du nombre d'octets utilisés dans le codage (uniquement les séquences valides)

Caractères codés	Représentation binaire UTF-8	Premier octet valide (hexadécimal)	Signification
U+0000 à U+007F	0xxxxxxx	00 à 7F	1 octet, codant 7 bits
U+0080 à U+07FF	110xxxxx 10xxxxxx	C2 à DF	2 octets, codant 11 bits
U+0800 à U+0FFF	1110**0000** 10xxxxxx 10xxxxxx	E0 (le 2e octet est restreint de A0 à BF)	3 octets, codant 16 bits
U+1000 à U+1FFF	1110**0001** 10xxxxxx 10xxxxxx	E1	
U+2000 à U+3FFF	1110**001x** 10xxxxxx 10xxxxxx	E2 à E3	
U+4000 à U+7FFF	1110**01xx** 10xxxxxx 10xxxxxx	E4 à E7	
U+8000 à U+BFFF	1110**10xx** 10xxxxxx 10xxxxxx	E8 à EB	
U+C000 à U+CFFF	1110**1100** 10xxxxxx 10xxxxxx	EC	
U+D000 à U+D7FF	1110**1101** 10xxxxxx 10xxxxxx	ED (le 2e octet est restreint de 80 à 9F)	
U+E000 à U+FFFF	1110**111x** 10xxxxxx 10xxxxxx	EE à EF	
U+10000 à U+1FFFF	11110**000** 1001xxxx 10xxxxxx 10xxxxxx	F0 (le 2e octet est restreint de 90 à BF)	4 octets, codant 21 bits
U+20000 à U+3FFFF	11110**000** 101xxxxx 10xxxxxx 10xxxxxx		
U+40000 à U+7FFFF	11110**001** 10xxxxxx 10xxxxxx 10xxxxxx	F1	
U+80000 à U+FFFFF	11110**01x** 10xxxxxx 10xxxxxx 10xxxxxx	F2 à F3	
U+100000 à U+10FFFF	11110**100** 1000xxxx 10xxxxxx 10xxxxxx	F4 (le 2e octet est restreint de 80 à 8F)	

Le principe fondateur de l'UTF-8 est qu'il est **adaptatif** : les caracères les plus fréquents sont codés sur un octet, qui est la taille minimale (et qui donne le 8 de "UTF-8"). Les autres caractères peuvent être codés sur 2, 3 ou 4 octets au maximum.

UTF-8 n'utilise que l'espace dont il a besoin pour un caractère. Cela signifie donc que certains caractères n'utilisent qu'un seul octet, et d'autres deux, trois et même quatre.
UTF-8 utilise les mêmes codes qu'**ASCII** pour les 127 premiers caractères, et se sert d'octets additionnels pour représenter des caractères spéciaux comme 'é'.

Par exemple, le caractère Z serait représenté de la même façon qu'en ASCII : `01011010`

Toutefois, le caractère ç devra être représenté en deux octets, car il ne fait pas partie des 127 caractères originaux.
L'encodage d'un caractère multi-octet se fait comme suit :

- les premiers bits identifient le nombre d'octets à utiliser.
- 0xxxxxxx : signifie que le caractère fait 1 octet de long
- 110xxxxx signifie que le caractère fait 3 octets de long,
- 1110xxxx signifie 3 octets,
- 11110xxx 4 octets,

On note U+XXXX un caractère encodé en UTF8. Les bits restants sont utilisés pour représenter le numéro du caractère.

Exemple :

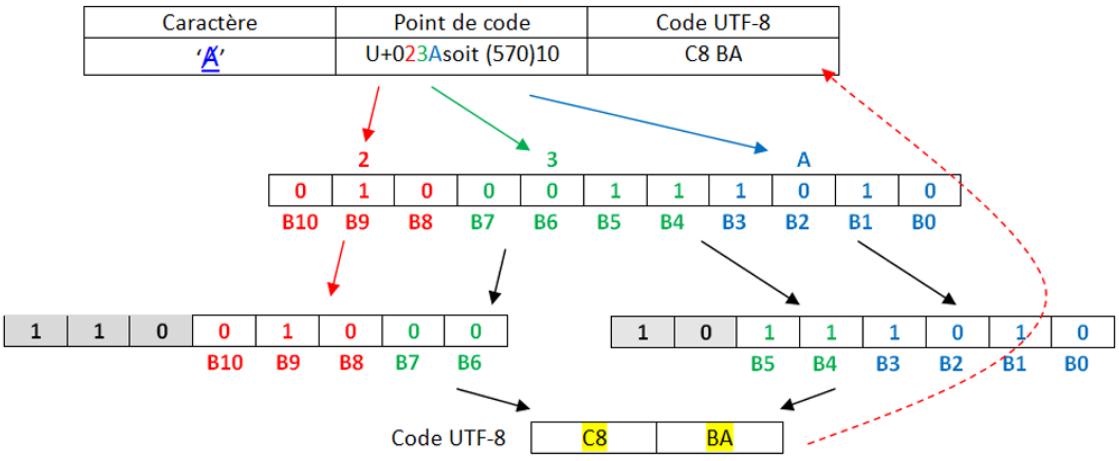
Latin étendu B

HEX		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	DEC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
180	384	ƀ	Ɓ	Ƃ	ƃ	Ƅ	ƅ	Ɔ	Ƈ	ƈ	Ɖ	Ɗ	Ƌ	ƌ	ƍ	Ǝ	Ə
190	400	Ɛ	Ƒ	ƒ	Ɠ	ƔƔ	ƕ	Ɩ	Ɨ	Ƙ	ƙ	ƚ	ƛ	Ɯ	Ɲ	ƞ	Ɵ
1A0	416	Ơ	ơ	Ƣ	ƣ	Ƥ	ƥ	Ʀ	Ƨ	ƨ	Σ	ƪ	ƫ	Ƭ	ƭ	Ʈ	Ư
1B0	432	ư	Ʊ	Ʋ	Ƴ	ƴ	Ƶ	ƶ	Ʒ	Ƹ	ƹ	ƺ	Ƶ	ƶ	ƷƷ	ƸƸ	ƹƹ
1C0	448	ǀ	ǁ	ǂ	ǃ	DŽ	Dž	dž	LJ	Lj	lj	NJ	Nj	nj	Ǎ	ǎ	Ǐ
1D0	464	ǐ	Ǒ	ǒ	Ǔ	ǔ	Ǖ	ǖ	Ǘ	ǘ	Ǚ	ǚ	Ǜ	ǜ	ǝ	Ǟ	ǟ
1E0	480	Ǡ	ǡ	Ǣ	ǣ	Ǥ	ǥ	Ǧ	ǧ	Ǩ	ǩ	Ǫ	ǫ	Ǭ	ǭ	Ǯ	ǯ
1F0	496	ǰ	DZ	Dz	dz	Ǵ	ǵ	Ƕ	Ƿ	Ǹ	ǹ	Ǻ	ǻ	Ǽ	ǽ	Ǿ	ǿ
200	512	Ȁ	ȁ	Ȃ	ȃ	Ȅ	ȅ	Ȇ	ȇ	Ȉ	ȉ	Ȋ	ȋ	Ȍ	ȍ	Ȏ	ȏ
210	528	Ȑ	ȑ	Ȓ	ȓ	Ȕ	ȕ	Ȗ	ȗ	Ș	ș	Ț	ț	Ȝ	ȝ	Ȟ	ȟ
220	544	Ƞ	ȡ	Ȣ	ȣ	Ȥ	ȥ	Ȧ	ȧ	Ȩ	ȩ	Ȫ	ȫ	Ȭ	ȭ	Ȯ	ȯ
230	560	Ȱ	ȱ	Ȳ	ȳ	ȴ	ȵ	ȶ	ȷ	ȸ	ȹ	Ⱥ	Ȼ	ȼ	Ƚ	Ⱦ	ɀ
240	576	ɂ	ɂ	ɔ	Ƀ	Ʉ	Ʌ	Ɇ	ɇ	Ɉ	ɉ	Ɋ	ɋ	Ɍ	ɍ	Ɏ	ɏ

A partir de l'extrait de la table Unicode(version hexa), rappelez le Point de Code du caractère Ⱥ (valeur numérique).

Le caractère appartient à l'intervalle U+0080 à U+07FF, donc les **11 bits** seront répartis sur deux octets en code UTF-8.

- Convertir le code en binaire sur 11 bits
- 11 bits sur deux octects selon la disposition spécifiée dans la norme



Exercice

HEX		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	DEC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
80	128	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
90	144	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
A0	160	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY -	®	¯
B0	176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0	192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	I	ï
F0	240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	Þ	ÿ

Faire de même avec la lettre é

Exercice

Quel est le code binaire de "défi" codé avec UTF-8 ?

Exercice

Quels mots se cachent sous les codes UTF-8 suivants ?

1. `01101000 01100101 01101100 01101100 01101111`
2. `01101001 01101110 01100110 01101111 01110010 01101101 01100001 01110100 01101001 01110001 01110101 01100101`
3. `01100010 01101001 01101110 01100001 01101001 01110010 01100101`

Exercice

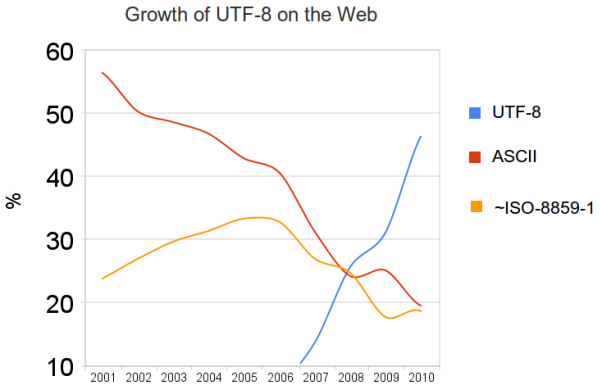
Le défi du cours : codage UTF-8, décoder le texte ci-dessous :

- soit en créant une fonction python
- soit à la main.

0101 0110 01101111 01101001 01101100 11000011 10100000 00100000
0011 0001 11100010 10000010 10101100 00101100 00100000 01110101
0110 1110 00100000 11110000 10011111 10011000 10000100 00100000
0110 0101 01110100 00100000 01101101 11000011 10101010 01101101
0110 0101 00100000 11110000 10011101 10000100 10011110 00100001

3.1 Utilisation grandissante de l'encodage UTF-8

La majorité des sites internet utilisent maintenant l'UTf-8, tout comme les systèmes d'exploitation récents.



4. Applications : Codage XOR

4.1 Extrait sujet BAC : Codage XOR

L'objectif de l'exercice est d'étudier une méthode de cryptage d'une chaîne de caractères à l'aide du codage ASCII et de la fonction logique XOR.

Question 1

Le nombre 65, donné ici en écriture décimale, s'écrit 01000001 en notation binaire. En détaillant la méthode utilisée, donner l'écriture binaire du nombre 89.

Question 2

La fonction logique **OU EXCLUSIF**, appelée **XOR** et représentée par le symbole ⊕, fournit une sortie égale à 1 si l'une ou l'autre des deux entrées vaut 1 mais pas les deux.

On donne ci-dessous la table de vérité de la fonction XOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Poser et calculer l'opération : 11001110 ⊕ 01101011

On donne, ci-dessous, un extrait de la table ASCII qui permet d'encoder les caractères de A à Z.
On peut alors considérer l'opération XOR entre deux caractères en effectuant le XOR entre les

codes ASCII des deux caractères.
Par exemple : 'F' XOR 'S' sera le résultat de 01000110 ⊕ 01010011.

Code ASCII Décimal	Code ASCII Binaire	Caractère
65	01000001	A
66	01000010	B
67	01000011	C
68	01000100	D
69	01000101	E
70	01000110	F
71	01000111	G
72	01001000	H
73	01001001	I
74	01001010	J
75	01001011	K
76	01001100	L
77	01001101	M

Code ASCII Décimal	Code ASCII Binaire	Caractère
78	01001110	N
79	01001111	O
80	01010000	P
81	01010001	Q
82	01010010	R
83	01010011	S
84	01010100	T
85	01010101	U
86	01010110	V
87	01010111	W
88	01011000	X
89	01011001	Y
90	01011010	Z

Le cryptage XOR est un système de cryptage basique mais pas trop limité. Ainsi, il a beaucoup été utilisé dans les débuts de l'informatique et continue à l'être encore aujourd'hui car il est facile à implémenter, dans toutes sortes de programmes.

Chaque caractère du message à coder est représenté par un entier, le code ASCII. Ce nombre est lui-même représenté en mémoire comme un nombre binaire à 8 chiffres (les bits). On choisit une clé que l'on place en dessous du message à coder, en la répétant autant de fois que nécessaire. Le message et la clé étant converti en binaire, on effectue un XOR, bit par bit. Le résultat en binaire peut être reconverti en caractères ASCII et donne alors le message codé.

Question 3.

Chiffrer **MESSAGE** avec la clé **CLE**. Pour cela recopier et compléter le tableau ci-dessous :

Lettre	M	E	S	S	A	G	E
Code ASCII							
Code binaire							
Clé	C	L	E	C	L	E	C
Code ASCII							
Clé binaire							

XOR								
Code ASCII								
LETTRE Codée								

```
A vous avec message = "SPECIALITE NSI"  et clef = "TERM"
```

Question 4.

Recopier et compléter la table de vérité de $(E1 \oplus E2) \oplus E2$.

E_1	E_2	$E_1 \oplus E_2$	$(E_1 \oplus E_2) \oplus E_2$
0	0	0	
0	1	1	
1	0	1	
1	1	0	

A l'aide de ce résultat, proposer une démarche pour décrypter un message crypté.

4.2 Approfondissement : Programmation du chiffremment XOR

Aide :

- On pourra utiliser la fonction native du langage Python ord(c) qui prend en paramètre un caractère c et qui renvoie un nombre représentant le code ASCII du caractère c
- `format(14, '08b')` donne l'ecriture binaire de 14 sous la forme '00001110'

Question préliminaire

1. Ecrire une fonction `xor(v1,v2)` où v1 et v2 sont soit 0 soit 1.
2. Ecrire une fonction `xorbinaire(n1,n2)` où n1 et n2 sont nombres binaires, cette fonction doit renoyer le résultat sous forme binaire.

Question 5.

Ecrire le corps de la fonction `convertit_texte_en_binaire(texte)` qui convertit la chaine de caractères ASCII texte passée en paramètre en une chaine binaire et retourne cette chaine binaire. Chaque caractère sera représenté par son code ASCII en binaire sur un octet.

Exemple : `convertit_texte_en_binaire("NSI")` doit retourner la chaine : `010011100101001101001001`

En effet :

Le code ASCII de "N" est 78 en décimal = 01001110 en binaire sur un octet
Le code ASCII de "S" est 83 en décimal = 01010011 en binaire sur un octet
Le code ASCII de "I" est 73 en décimal = 01001001 en binaire sur un octet
Et, '01001110' + '01010011' + '01001001' = '010011100101001101001001'

Question 6.

Ecrire le corps de la fonction `convertit_binaire_vers_entier_base_10(chaine_binaire)` qui convertit la chaine binaire chaine_binaire passée en paramètre en le nombre décimal correspondant et retourne ce nombre décimal.

Exemple : `convertit_binaire_vers_entier_base_10("01001110")` doit retourner l'entier 78

En effet : 01001110 en base 2 = 0x1 + 1x2 + 1x4 + 1x8 + 0x16 + 0x32 + 1x64 + 0x128 = 2 + 4 + 8 + 64 = 78

Question 7.

Compléter la fonction `convertit_binaire_en_texte(chaine_binaire)` qui convertit la chaine chaine_binaire passée en paramètre composée d'octets binaires représentant des caractères codés en ASCII en une chaine de caractères et retourne cette chaine de caractères.

```
def convertit_binaire_en_texte(chaine_binaire):
decoupe=[]
texte=''
compteur=1
lettre=''
for v in chaine_binaire:
    if compteur==...:
        lettre+=...
        decoupe.append(...)
        lettre=...
        compteur=...
    else:
        lettre+=...
        compteur+=...
for tab in decoupe:

texte=chr(convertit_binaire_vers_entier_base_10(tab))+texte
return texte
```

Exemple : `convertit_binaire_en_texte("010011100101001101001001")` doit retourner la chaine : 'NSI'.

En effet :

'010011100101001101001001' = '01001110' + '01010011' + '01001001'

L'octet binaire 01001110 correspond au nombre décimal 78 qui représente en ASCII le caractère "N"
L'octet binaire 01010011 correspond au nombre décimal 83 qui représente en ASCII le caractère "S"
L'octet binaire 01001001 correspond au nombre décimal 73 qui représente en ASCII le caractère "I"
La chaîne retournée est donc bien "NSI".

Question 8.

Ecrire une fonction `def reperter_cle(chaine,clef)` permettant de compléter la clé afin d'avoir la même longuer que le message.
Exemple : `reperter_cle('message','cle')` doit retourner 'cleclec'

Question 9.

Ecrire le corps de la fonction `chiffre_xor(chaine,clef)` qui chiffre la chaine `chaine` passée en paramètre avec la clef `clef` passée en paramètre en effectuant l'opération XOR bit à bit entre la chaine binaire et la clef binaire (répétée). La fonction doit retourner la chaine ainsi chiffrée.

Exemple : Chiffrons la chaine "SPECIALITE NSI" avec la clef "TERM".

- `convertit_texte_en_binaire("SPECIALITE NSI")` retourne `010100110101000001000101010000110100100101000001 010011000100100101010100010001010010000001001110 0101001101001001`.
- `convertit_texte_en_binaire("TERM")` retourne `01010100010001010101001001001101`.
- `chiffre_xor("SPECIALITE NSI" ,'TERM`) doit retourner `00000111000101010001011100001110000111010000010000011111 00000010000000000000000000111001000000011000001110000110 0`.

Question 10.

Ecrire le corps de la fonction `texte_xor(chaine,clef)` qui doit retourner la chaine ainsi chiffrée sous forme de texte.