

Représentation des caractères

Crédits : Ce cours est presque intégralement repris du cours proposé par Gilles Lassus sur le [Codage des caractères](#), diffusé sous licence CC BY-SA

Comme nous l'avons déjà vu, une machine ne comprend que le langage binaire. Ainsi, même les textes, et donc les caractères doivent être représentés par des mots de 0 et de 1. Nous allons voir comment dans ce document !

■ Au départ l'ASCII

Avant 1960, il existait de nombreux systèmes pour coder les caractères, souvent incompatibles entre eux.

En **1960**, l'organisation internationale de normalisation (ISO) décide de créer la **norme ASCII** (pour *American Standard Code for Information Interchange*) pour uniformiser le codage des caractères.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Fig. 1 - Table ASCII.

Crédit : ASCII-Table.svg: ZTZ32derivative work: Usha, Public domain, via Wikimedia Commons

En ASCII, 127 « *points de code* » (= nombres associés aux caractères) sont disponibles. Les caractères sont donc codés sur **7 bits**.

Par exemple, le caractère **A** correspond à la valeur décimale 65 ou à la valeur hexadécimale 41. Ainsi, il est représenté en binaire sur 7 bits par `1000001`.

```
>>> bin(65) # conversion en binaire
'0b1000001'
```

La fonction `chr` renvoie le caractère correspondant à la valeur décimale passée en paramètre :

```
>>> chr(78)
'N'
>>> chr(123)
'{'
```

Si on dispose de la valeur binaire d'un caractère codé en ASCII, on peut trouver ce caractère facilement en utilisant `int` puis `chr` :


```
>>> int('1100101', 2) # conversion en décimal
101
>>> chr(101)
'e'
```

Ou directement :

```
>>> chr(int('1100101', 2))
'e'
```

Exercice 1

 **Question 1 :** Utilisez les fonctions `int`, `chr` pour déterminer le caractère codé `1010101` en ASCII.

 **Question 2 :** En vous aidant de la fonction `bin` vérifiez la cohérence avec la table ASCII donnée au-dessus. Expliquez.

Exercice 2 : Décoder un message codé en ASCII

On dispose d'un message `msg` écrit en ASCII et on souhaite décoder ce message.

```
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
```

Comme ce serait long de le faire à la main caractère par caractère, on va écrire un programme qui permet de le faire.

Mais avant cela, une présentation de la méthode `split` qui s'applique à une chaîne de caractères et qui sera utile pour la suite.

La méthode `split` permet de décomposer une chaîne de caractères en une liste en utilisant le caractère séparateur passé en paramètre :

```
>>> ch1 = "une phrase d'exemple"
>>> ch1.split('e')
['un', ' phras', " d'", 'x', 'mpl', '']
>>> ch2 = "pou, caillou, genou, chou, hibou, joujou, bijou"
>>> ch2.split(',')
['pou', ' caillou', ' genou', ' chou', ' hibou', ' joujou', ' bijou']
>>> ch2.split(', ') # observez bien la différence avec l'instruction précédente
['pou', 'caillou', 'genou', 'chou', 'hibou', 'joujou', 'bijou']
```

 **Question :** Complétez le code ci-dessous qui permet de décoder notre message.

L'idée est de convertir le message binaire en une liste de codes binaires puis de parcourir les différents codes binaires de cette liste pour les convertir en caractères et construire la chaîne de caractères qui decode ce message.

```
msg = "1101100 1100101 1110011 100000 1001110 1010011 1001001 100000 1100011 100111 1100101 1110011 1110100 100000
liste_codes_ascii = msg.split(...)
msg_decode = ""
for code in ...:
    caractere = ...
    msg_decode = msg_decode + caractere
print(msg_decode)
```

■ Par la suite, d'autres encodages

La norme ASCII convient bien à la langue anglaise, mais lorsque d'autres personnes que des américains ou des anglais ont voulu s'échanger des données textuelles, certains caractères étaient manquants : é, è, à, ñ, Ø, Ö, ß, 漢, ...

Les 7 bits de la table ASCII étaient insuffisants pour ajouter ces nouveaux caractères, il a donc été décidé de coder les caractères sur 8 bits pour arriver à ... 256 caractères.

C'est ainsi que de nouvelles tables, codant les caractères sur 8 bits, virent le jour.

En 1986, la table **ISO 8859-1**, aussi appelée **Latin-1**, a vu le jour et était principalement utilisée en Europe car elle ajoutait aux caractères de la table ASCII les caractères de l'alphabet du "latin".

Mais il manquait encore des caractères et après de nombreuses modifications successives (la dernière en date rajoutant par exemple le symbole €), on a aboutit à la célèbre table **ISO 8859-15**, appelée 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	non utilisé															
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	non utilisé															
Ax		ı	ç	£	€	¥	Š	š	Š	©	®	«	»	®	™	ˆ
Bx	°	±	²	³	Ž	μ	¶	·	ž	ı	º	»	Œ	œ	ÿ	ı
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

Fig. 2 - Table ISO/CEI 8859-15.

Source : article [ISO/CEI 8859-15](#) sur Wikipédia.

Utilisation :

Les codes sont donnés en hexadécimal :

- le caractère € correspond au code hexadécimal A4 (intersection de la ligne Ax et de la colonne x4), 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.



Rétro-compatibilité avec l'ASCII

Si vous regardez les 128 premiers caractères de cette table, de 00 à 7F en hexadécimal (soit de 0 à 127 en décimal), on retrouve exactement les caractères de la table ASCII. Cela permet à la table ISO 8859-15 d'être compatible avec la table ASCII et ainsi pouvoir décoder sans erreur des textes encodés préalablement avec la table ASCII.

Exemple :

Le fichier test.txt contient le texte Ça marche très bien ! enregistré 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, il ne contient que les données et rien d'autre.)

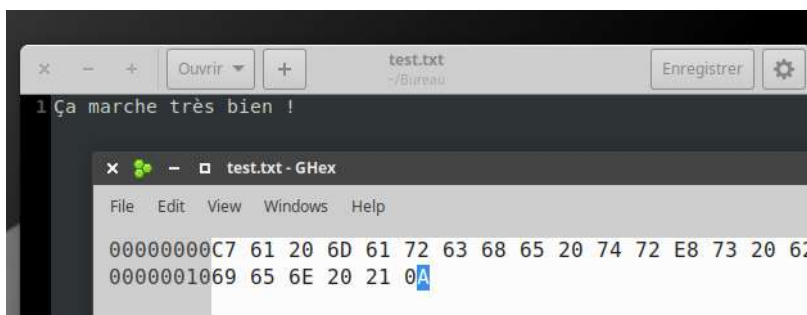


Fig. 3 - Ouverture d'un fichier avec un éditeur hexadécimal.

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	Inutilisé															
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	Inutilisé															
Ax	NBSP	'	´	£	€	Ɔp	ı	§	¨	©	ı	«	¬	SHY		—
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	Ω
Cx	ı	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	ı	ÿ	á	é	ή	ı	
Ex	ύ	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	ı	ü	ó	ú	ώ	

Fig. 4 - Table ISO/CEI 8859-7.
Source : article ISO/CEI 8859-7 sur Wikipédia.

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	Inutilisé															
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	Inutilisé															
Ax	NBSP	ก	ข	ค	ด	ต	ถ	ท	ด	น	บ	ป	ผ	ฝ	ภ	จ
Bx	จ	ช	ฌ	ฉ	ช	น	น	น	น	น	น	น	น	น	น	น
Cx	ก	ข	ค	ด	ต	ถ	ท	ด	น	บ	ป	ผ	ฝ	ภ	จ	ช
Dx	ช	ฌ	ฉ	ช	น	น	น	น	น	น	น	น	น	น	น	น
Ex	เ	แ	ไ	ใ	ใ	ใ	ใ	ใ	ใ	ใ	ใ	ใ	ใ	ใ	ใ	ใ
Fx	อ	อ	บ	บ	ค	ค	ค	ค	ค	ค	ค	ค	ค	ค	ค	ค

Fig. 5 - Table ISO/CEI 8859-11.
Source : article ISO/CEI 8859-11 sur Wikipédia.

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

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

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



Fig. 6 - Problème d'affichage dans un navigateur.



L'exercice 5 permettra d'expliquer ce genre d'affichages étranges. Mais il faut d'abord aborder l'encodage UTF-8, le plus utilisé actuellement (et de loin).

Normalement, pour un navigateur, une page web correctement codée doit contenir dans une balise `meta` le `charset` utilisé (*charset* signifie "jeu de caractères"), comme sur la ligne 4 ci-dessous :

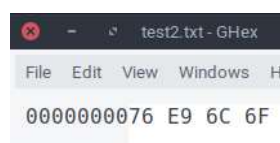
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>

</body>
</html>
```

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

Exercice 3

On a ouvert un fichier texte avec un éditeur hexadécimal comme le montre la capture d'écran ci-dessous.



Question : Le mot représenté par les octets ci-dessous est-il encodé en ASCII ou en Latin-9 ? Quel est ce mot ? Expliquez.

■ Enfin une normalisation avec 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.



Slogan du Consortium Unicode

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

✎ Exercice 4

🔖 **Question** : Donnez la traduction de ce slogan.

Le consortium crée 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 qui est de loin le plus utilisé.

UTF-8

Définition du nombre d'octets utilisés dans le codage (attention ce tableau de principe contient des séquences non valides)

Caractères codés	Représentation binaire UTF-8	Premier octet valide (hexadécimal)	Signification
U+0000 à U+007F	0bbb·bbbb	00 à 7F	1 octet, codant jusqu'à 7 bits
U+0080 à U+07FF	110b·bbbb 10bb·bbbb	C2 à DF	2 octets, codant jusqu'à 11 bits
U+0800 à U+FFFF	1110·bbbb 10bb·bbbb 10bb·bbbb	E0 à EF	3 octets, codant jusqu'à 16 bits
U+10000 à U+10FFFF	1111·000b 10bb·bbbb 10bb·bbbb 10bb·bbbb	F0 à F3	4 octets, codant jusqu'à 21 bits
	1111·0100 10bb·bbbb 10bb·bbbb 10bb·bbbb	F4	

Fig. 7 - Codage des caractères en UTF-8.

Source : article [UTF-8](#) de Wikipedia.

Le principe fondateur de l'UTF-8 est qu'il est **adaptatif** : les caractères 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.



Rétro-compatibilité avec l'ASCII

Les caractères de 0000 à 007F en hexadécimal sont représentés sur 1 octet (codant 7 bits) et correspondent aux caractères de la table ASCII, ce qui assure la rétro-compatibilité comme

évoquée plus haut. C'est la première ligne du tableau ci-dessus.

Exemple du codage en UTF-8 du caractère « € »

- On peut utiliser le site <https://symbl.cc/fr/> pour trouver le nombre Unicode du caractère « € ».
- En tapant « € » dans la barre de recherche on trouve que son nombre Unicode est U+20AC.
- On repère dans le tableau sur combien d'octets se code le nombre U+20AC :

Définition du nombre d'octets utilisés dans le codage (attention ce tableau de principe contient des séquences non valides)

Caractères codés	Représentation binaire UTF-8	Premier octet valide (hexadécimal)	Signification
U+0000 à U+007F	0bbb·bbbb	00 à 7F	1 octet, codant jusqu'à 7 bits
U+0080 à U+07FF	110b·bbbb 10bb·bbbb	C2 à DF	2 octets, codant jusqu'à 11 bits
U+0800 à U+FFFF	1110·bbbb 10bb·bbbb 10bb·bbbb	E0 à EF	3 octets, codant jusqu'à 16 bits
U+10000 à U+10FFFF	1111·000b 10bb·bbbb 10bb·bbbb 10bb·bbbb	F0 à F3	4 octets, codant jusqu'à 21 bits
	1111·0100 1000·bbbb 10bb·bbbb 10bb·bbbb	F4	

- U+20AC correspond à la troisième ligne car en hexadécimal 20AC est compris entre 0800 et FFFF : il faudra donc 3 octets pour coder le symbole « € » en UTF-8.
- Les 16 bits de son code binaire seront répartis sur les 3 octets aux emplacements en vert donnés au-dessus, en complétant ces 16 bits avec des zéros à gauche si nécessaire.
- On convertit alors la valeur hexadécimale 20AC en binaire :

```
>>> int('20AC', 16) # conversion d'hexadécimal (base 16) en décimal (base 10)
8364
>>> bin(8364) # conversion en binaire (base 2)
'0b10000010101100'
```

- On trouve que 20AC se code 10000010101100 en binaire, soit sur 14 bits. Il faudra donc rajouter deux zéros à gauche pour arriver à 16 bits : 0010000010101100
- On encapsule ces 16 bits aux emplacements en vert (0010, puis 000010 et enfin 101100) et on obtient alors :

```
11100010 10000010 10101100
```

- **Conclusion** : le symbole « € » se code 11100010 10000010 10101100 en UTF-8.

Exercice 5 : La réponse à une question existentielle

La fonction `ord` de Python permet de connaître la valeur décimale en UTF-8 d'un caractère :

```
>>> ord("€") # renvoie 8364 comme vu plus haut
8364
```

Question 1 : Utilisez les fonctions `ord` puis `bin` pour écrire en binaire le nombre associé au caractère `é` en UTF-8.

Question 2 : En vous aidant de l'exemple de l'encodage du caractère « € » en UTF-8, déterminez l'encodage du caractère `é` en UTF-8. *Détaillez la démarche.*

Question 3 : Convertissez les deux octets obtenus (grâce à `int`) puis en hexadécimal (grâce à `hex`).

Question 4 : Si un logiciel considère à tort que les deux octets servant à encoder le `é` en UTF-8 servent à encoder deux caractères en ISO 8859-15, quels seront ces deux caractères ? *Expliquez.*

Question 5 (bilan) : Expliquez en quelques lignes, pourquoi certains caractères s'affichent parfois de façon étrange lorsque que l'on ouvre certains documents comme on le voit dans la capture d'écran du paragraphe "Que fait un logiciel à l'ouverture d'un fichier texte ?".

L'UTF-8 l'emporte aujourd'hui très largement

Désormais, la majorité des sites Web utilisent maintenant l'UTF-8, tout comme les systèmes d'exploitation récents.

De 2001 à 2010 :

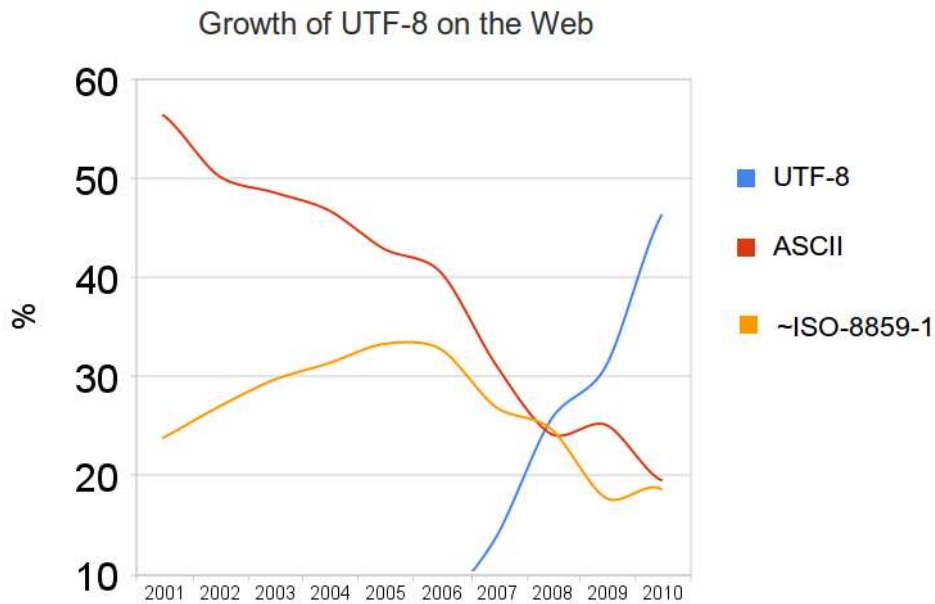


Fig. 8 - Évolution de l'utilisation de l'UTF-8 sur le Web entre 2001 et 2010.
 Crédit : Krauss, CC BY-SA 4.0, via Wikimedia Commons

De 2012 à 2022 :

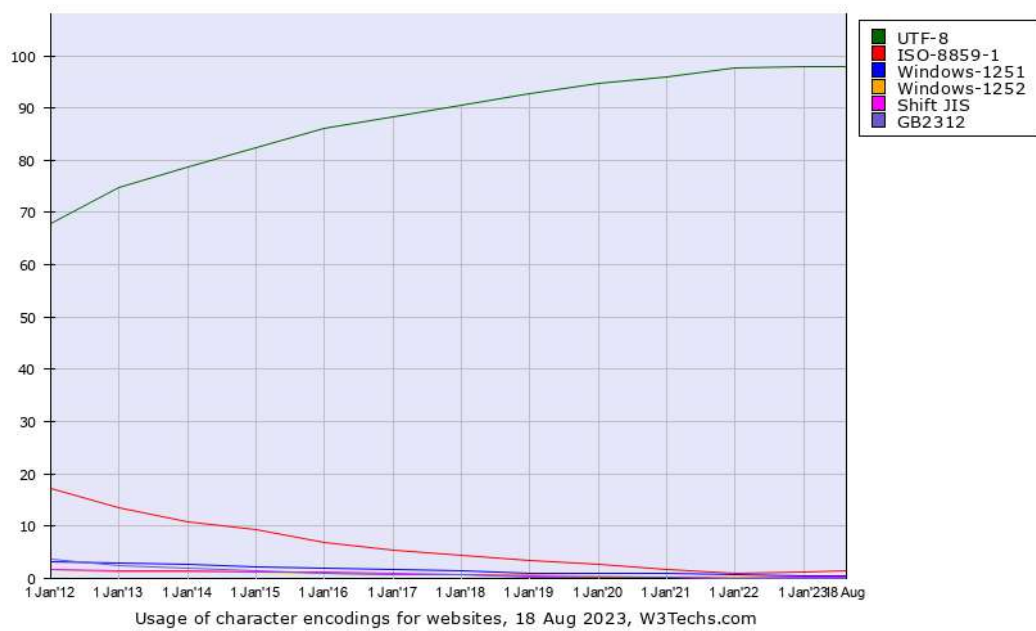


Fig. 9 - Évolution de l'utilisation de l'UTF-8 sur le Web entre 2012 et 2022.
 Source : https://w3techs.com/technologies/history_overview/character_encoding/ms/y.

■ Encodage avec Python

Grâce à la fonction `open` on peut définir l'encodage utilisé pour *lire* ou *écrire* dans un fichier avec Python.

Le fichier texte `mon_fichier_utf8.txt` encodé en UTF-8 contient la phrase :

Ce pull est très cher, il coûte 100 €.

On va l'ouvrir pour en lire le contenu avec Python.

Si on l'ouvre en définissant le bon encodage, grâce au paramètre `encoding`, il n'y a aucun problème.

```
>>> f = open('fichier_utf8.txt', 'r', encoding='utf-8') # 'r' pour read (lecture)
>>> contenu = f.read() # lecture
>>> f.close() # fermeture du flux de lecture
```



```
>>> print(contenu)
Ce pull est très cher, il coûte 100 €.
```

En revanche, si on le lit avec le mauvais encodage, par exemple Latin-9, on obtient les problèmes évoqués plus haut :

```
>>> f = open('fichier_utf8.txt', 'r', encoding='latin9') # MAUVAIS ENCODAGE
>>> contenu = f.read() # lecture
>>> f.close() # fermeture du flux de lecture
```

```
>>> print(contenu)
Ce pull est trÃs cher, il coÃte 100 -
```

Lorsqu'on veut écrire dans un fichier, on peut aussi définir l'encodage que l'on utilisera (toujours grâce au paramètre `encoding`). Par exemple, ici on ouvre un fichier `fichier_latin9.txt` (le fichier est créé car n'existe pas) en mode écriture en utilisant la table Latin-9 :

```
>>> f2 = open('fichier_latin9.txt', 'w', encoding='latin9') # 'w' pour write (écriture)
>>> f2.write("Représentation d'un texte en machine") # écriture
>>> f2.close()
```

Si on lit ce fichier nouvellement créé en définissant le bon encodage (celui utilisé pour l'écriture), tout se passe bien :

```
>>> f2 = open('fichier_latin9.txt', 'r', encoding='latin9')
>>> contenu = f2.read()
>>> f2.close()
```

```
>>> print(contenu)
Représentation d'un texte en machine
```

Mais si on le lit en définissant l'encodage UTF-8, il y a un problème :

```
>>> f2 = open('fichier_latin9.txt', 'r', encoding='utf-8')
>>> contenu = f2.read()
>>> f2.close()
```

```
>>> print(contenu) # c'est le 'é' qui pose problème au décodage
-----
UnicodeDecodeError                                Traceback (most recent call last)
<ipython-input-11-138de5bdad86> in <module>
      1 f2 = open('fichier_latin9.txt', 'r', encoding='utf-8')
----> 2 contenu = f2.read()
      3 print(contenu)
      4 f2.close()

D:\Anaconda\lib\codecs.py in decode(self, input, final)
    320     # decode input (taking the buffer into account)
    321     data = self.buffer + input
--> 322     (result, consumed) = self._buffer_decode(data, self.errors, final)
    323     # keep undecoded input until the next call
    324     self.buffer = data[consumed:]

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 4: invalid continuation byte
```

Exercice 6 (Bonus)

En vous aidant de ce qui vient d'être dit, écrivez un programme Python qui permet de convertir le fichier `fichier_latin9.txt` (encodé en Latin-9) créé juste au-dessus en un fichier `fichier_converti.txt` encodé en UTF-8.

■ Bilan

- De nombreux systèmes ont existé au cours du temps pour représenter les caractères : le premier qui a tenté d'uniformiser l'encodage des caractères était la norme **ASCII**.
- Cette norme ne permettait de coder que 127 caractères, ce qui s'est vite révélé très insuffisant pour d'autres langues que l'anglais.

- D'autres normes ont ensuite vu le jour, chacune s'appuyant sur une **table de caractères** qui donnait la correspondance entre caractère et représentation binaire (ou hexadécimale). Chacune d'elle était *rétro-compatible* avec l'ASCII de manière à pouvoir décoder des textes préalablement encodés en ASCII.
- Finalement, c'est la norme **Unicode**, avec son encodage **UTF-8** créé en 1996, qui est venu réunir tous les caractères dans une immense table (il y a encore beaucoup de places pour de nouveaux caractères, cela a permis d'y intégrer les émojis par exemple).
- L'UTF-8 a la particularité de coder les caractères sur un nombre d'octets variant de 1 à 4 : les plus courants sont codés sur 1 octet, les moins courants sur davantage. C'est désormais l'encodage à utiliser pour pouvoir permettre les échanges au niveau mondial dans toutes les langues sans problèmes.
- Comme plusieurs encodages ont co-existé pendant longtemps, même si cela devient plus rare désormais, il n'est pas impossible que certains logiciels détectent mal l'encodage utilisé pour décoder un (ancien) document : c'est ce qui conduit encore à des affichages absurdes sur nos écrans !

Références

- Le cours de Gilles Lassus sur le [Codage des caractères](#), diffusé sous licence CC BY-SA.
- Cours du DIU EIL, université de Nantes, Christophe Declercq.

Germain Becker, Lycée Emmanuel Mounier, Angers.



Voir en ligne : info-mounier.fr/premiere_nsi/types_base/caracteres