Texte et fichiers

Benjamin Monmege et François Brucker

Texte

Tout est nombre

- Codage = transformer une information en nombre
- Base = octet, équivalent à un byte, 8 bits, 0xFF en hexadécimal, 255 en décimal

Différents codages pour le texte

- ASCII
- ISO-8859-1
- unicode/utf

Code ASCII

- chaque symbole codé sur 7 bits

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	[ENG 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]

Extension : code ISO-8859-1, latin1

- chaque symbole codé sur un octet (8 bits)
- permet d'écrire en français, les accents y étant présent
- 191 caractères imprimables

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
 ¡¢£¤¥¦§¨©ª«¬®¯
°±²³´µ¶·¸¹º»¼½¾¿
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ
ÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß
àáâãäåæçèéêëìíîï
ðñòóôõö÷øùúûüýþÿ
```

Recommandé actuellement : unicode

- permet d'associer à tous les symboles possibles un nombre
- voir https://unicode-table.com/fr/# et scrollez vers le bas pour voir tous les symboles unicodes possibles
- chaque symbole est codé sur 2 ou 3 octets et noté `U+XXXXXX`
- symboles rangés en *plans* de 2 octets (0x0000 à 0xFFFF en hexadécimal)
  - 4 chiffres pour le premier plan, appelé plan multilingue de base (donc entre U+0000 et U+FFFF)
  - 5 chiffres pour les 15 plans suivants (entre U+10000 et U+FFFFF)
  - 6 chiffres pour le dernier plan (entre U+100000 et U+10FFFF)

Codage utf

- codage unicode intenable en pratique, car coder tout caractère sur 3 octets prendrait trop de place
- émergence des formats *utf* (unicode transformation format), dont utf-8
  - chaque symbole sur 1, 2, 3 ou 4 octets
  - un octet de plus que tout nombre unicode au maximum
  - mais le plus souvent (si on écrit en anglais en particulier) tout est stocké sur 1 octet par caractère, comme l'ascii

Exemple du é

- code unicode : `0xE9 = 233`. Python : `ord('é')`
- code utf-8 : 2 octets `0xC3` et `0xA9`. Python : `'é'.encode('utf8')`
- en binaire: `bin(0xC3)` rend '0b11000011' et `bin(0xA9)` rend '0b10101001'
- en regardant le code utf-8 sur 2 octets https://fr.wikipedia.org/wiki/UTF-8#Description, on voit que le code unicode est la concaténation des 5 derniers bits du premier octet (qui commence par 110) et des 6 dernier bits du deuxième (qui commence par 10). On obtient ainsi `00011101001` qui est : `int('00011101001', 2)` et vaut : 233. Ouf, la boucle est bouclée on retrouve bien le code unicode de é.

Fichiers

Système de fichiers

- Fichier = suite de blocs sur le disque dur
- Chaque bloc a une adresse contenant le bloc suivant.
- liste chaînée de tableaux de même taille contenant les données (des octets)

Ce format a été choisi parce que :

- des fichiers de tailles différentes doivent pouvoir être ajoutés et supprimés du disque dur. La place y est donc fragmentée, et il n'est pas sur de pouvoir avoir la taille requise pour un fichier.
- on doit pouvoir ajouter des choses à un fichier sans avoir à tout re-écrire

Limitations

- on accède au fichier petit à petit. Il faut une *tête de lecture* qui voyage de bloc en bloc.
- il est impossible d'insérer des choses dans un fichier. On peut juste ajouter des éléments à la fin de celui-ci.
- le système de fichier dépend du système d'exploitation.
- écrire/lire sur un disque est coûteux en temps. Il est nécessaire d'avoir une mémoire tampon ce qui rend asynchrone la lecture et l'écriture d'un fichier.

Ce qu'on peut faire avec un fichier

- **ouvrir** le fichier : c'est se préparer à l'utiliser. Cette étape crée un *buffer* (mémoire tampon), un pointeur de bloc, une tête de lecture, etc.
- **fermer** un fichier : arrêter de s'en servir. Il est **indispensable** de toujours fermer un fichier après s'en être servi. On écrit en effet à cette étape les dernières instruction non encore passée du buffer au disque dur (c'est comme démonter une clé USB proprement).
- **lire** un fichier : on fait avancer la *tête de lecture* du fichier de tout le fichier, d'une ligne ou d'un nombre donné d'octets
- **écrire** un fichier : on ajoute des données à la fin d'un fichier (qui peut être initialement vide). Souvent on écrit pas tout de suite sur le disque dur, on attend d'avoir un nombre suffisant de données dans la mémoire tampon.

Types de fichiers

- fichiers texte
- fichiers binaires : nécessite un outil spécial pour les utiliser, c'est à dire un moyen de passer de l'octet à sa signification. Peuvent stocker des images, des vidéos, des programmes... Extension diverses possibles.
  - Passer des octets à leur signification pour un fichier se fait via un codec (codeur/décodeur).
  - Codec MPEG4 par exemple pour les vidéos
  - Il y a aussi des "codecs" pour les fichiers texte même si dans ce cas là on parlera plutôt d'encodage, comme utf-8...

Les fichiers en Python

- fichiers utilisés *via* un objet `file`
- objet créé par la commande `open` qui ouvre un fichier
- deux paramètres obligatoires :
  - le nom du fichier,
  - la façon dont on veut l'ouvrir :
    - `'r'` : en lecture. Tête de lecture placée au début du fichier
    - `'w'` : en écriture. Tête d'écriture placée au début du fichier. Donc **si le fichier contenait déjà des choses elles sont supprimées**
    - `'a'` : en écriture à la fin du fichier. Tête d'écriture placée à la fin du fichier. Donc si le fichier contenait déjà des choses elles ne sont **pas** supprimées

Manipulation des fichiers

- en lecture avec les méthodes `read` ou encore `readline` si c'est un fichier texte
- en écriture avec `write`
- fermeture de fichier avec la méthode `close`

Toutes les méthodes de fichiers sont décrites dans le module io (pour In et Out)

Adresse d'un fichier

- Par défaut python va chercher les fichiers à ouvrir dans le dossier du fichier python entrain d'être exécuté.
- Chercher un fichier dans un dossier spécifique : module pathlib

Exemple avec un fichier texte

Supposons que j'ai un fichier texte nommé `haiku.txt` contenant :

```
dans le vieil étang,
une grenouille saute,
un ploc dans l'eau.

Bashô.
```

Lecture du fichier en entier

```
f = open("haiku.txt", "r")  # ouverture d'un fichier text
poeme = f.read()
f.close()
print(poeme)
```

Nota Bene : Ne confondez pas le nom du fichier et son contenu. Le nom du fichier, ici `Haiku.txt`, nous permet de l'ouvrir en lecture grace à la commande `open`. Son contenu est ensuite mis dans la variable `poeme` grace à la méthode `read`.

Lecture ligne à ligne

```
f = open("haiku.txt", "r")
for ligne in f:  # boucle sur les lignes
    print(ligne)
f.close()
```

Nota Bene : Notez la ligne vide vide entre deux affichages. Ceci est dû au fait que chaque ligne du fichier contient déjà un retour à la ligne. Plus celui qui est ajouté automatiquement à la fin de l'instruction `python`.

Ajout au fichier

```
f = open("haiku.txt", "a")
f.write("\n")
f.write("1644-1694")
f.close()
```

On ajoute un retour à la ligne, puis les dates de naissance et de mort de Bashô.

Ecriture du fichier

```
f = open("haiku.txt", "w")
f.write("Noël est aux portes\n")
f.write("les dindes et les pintades\n")
f.write("rentrent dans les fours")
f.write("\n")
f.write("Salim Bellen")
f.close()
```

Une fois ouvert le fichier en écriture, tout son contenu précédent est perdu.

Utilisation de `with`

Vous verrez parfois l'utilisation du mot clé python `with` qui permet d'écrire :

```
with open("haiku.txt", "a") as f:
    f.write("\n")
    f.write("1644-1694")
```

- au début du bloc `with` le résultat de l'ouverture du fichier est appelé `f`
- A la fin du bloc `with` on ferme `f`
- s'il y a des erreurs, c'est également le bloc `with` qui s'en occupe pour nous.

Fichiers binaires

- Pour ouvrir un fichier binaire sous python, on utilise le caractère `'b'` suivi de ce que l'on veut faire avec.
- L'exemple suivant ouvre le fichier `haiku.txt` en lecture sous la forme d'un fichier binaire, puis place le contenu de celui-ci dans la variable `x`

```
f = open("haiku.txt", "br")
x = f.read()
```

- `type(x)` : `bytes` (une suite d'octets)
- `print(x)` : `b"dans le vieil \xc3\xa9tang,\nune grenouille saute,\nun ploc dans l'eau.\n\nBash\xc3\xb4.\n"`
- affichage des suites d'octets utilisant ascii + hexadécimal

```
texte = x.decode("utf-8")
```