La question de la représentation des textes se pose dans les mêmes termes que celles des entiers et des réels : trouver un **encodage** permettant d'écrire chaque symbole textuel sous la forme d'une succession de 0 et de 1.

Cependant, contrairement aux précédentes où une certaine structure pouvait être dégagée grâce à l'écriture binaire des nombres, le choix d'un tel encodage est nécessairement plus arbitraire car, a priori, peu de chose relie un caractère donné à un autre.

De plus, les différentes régions linguistiques du monde n'utilisent pas systématiquement les mêmes caractères. On peut penser aux différents alphabets (latin, cyrillique, arabe, chinois, etc.) mais aussi aux variations locales de ces alphabets (les accents par exemple).

Ces difficultés se sont historiquement illustrées par l'existence de divers **formats de textes**, différents encodages (parfois pour représenter le même alphabet) rendant difficile la conversion des textes informatiques d'une région à l'autre.

1 Codage ASCII

1.1 Généralités

Le codage ASCII (*American Standard Code for Information Interchange*) est un encodage proposé dans les années 60 par l'ANSI (*American National Standard Institute*) dans le but d'unifier l'ensemble des formats anglophones. Il représente 128 caractères encodés sur 8 bits (un octet).

Ces caractères sont les symboles de l'alphabet latin non accentués minuscules et majuscules (`a, ..., z, A, ..., Z`), les chiffres arabes (`0, ..., 9`), les symboles de ponctuation (virgule, point, parenthèses, etc.), les opérateurs arithmétiques (`+, -`, etc.), les caractères spéciaux (espace, retour à la ligne, tabulation, nouvelle page, etc.) et des caractères de contrôle non-imprimables.

L'association caractères / mots binaires 8 bits est donnée par la **table ASCII** qu'on peut trouer par exemple sur Wikipedia.

Bit de parité : Comme on l'a dit plus haut, la table ASCII permet d'encoder 128 caractères sur 8 bits. Cela peut paraître étrange car $128 = 2^7$ donc 7 bits suffisent. En réalité, la table ASCII met bien en oeuvre une correspondance 7 bits entre caractères et mots binaires mais un 8-ème bit (de poids fort, à gauche) est ajouté de sorte que le nombre de 1 du code soit toujours pair.

Par exemple le code 7 bit `010 1100` devient sur 8 bits `1010 1100` (pour avoir quatre 1) et `111 0111` devient `0111 0111`.

Ce bit, dit de parité, permet de détecter **certaines erreurs de transmission** car si un bit change parmi les huit bits du code, le nombre de 1 ne sera plus pair et on saura qu'il y a eu une erreur. En revanche, ce bit de parité n'est pas considéré comme faisant parti du code.

1.2 ASCII en Python

En Python, on sait déjà que les textes sont représentés entre guillemets ou entre apostrophes. Chaque symbole entre guillemets ou apostrophe est encodé en ASCII (en fait en UTF-8 comme on le verra plus tard).

Les fonctions Python permettant de passer d'un caractère à un code ASCII et inversement sont les suivantes :

— `ord(c)` renvoie le code du caractère `c` sous la forme d'un entier en base dix.

— `chr(n)` renvoie le caractère correspondant au code représenté par l'entier `n`.

De plus, dans une chaîne de caractère, il est possible de saisir les caractères via leur code ASCII en hexadécimal via la notation `\xhh` où `hh` désigne le code hexadécimal du caractère voulu.

Par exemple la chaîne `"\x43e\x63i es\x74..."` correspond à la chaîne `"Ceci est..."`. Cette technique pour désigner les caractères est appelée **caractères échappés**.

Il existe également des **raccourcis** permettant de saisir un certain nombre de caractères spéciaux. En voici des exemples couramment utilisés :

— `\n` : LF (nouvelle ligne) ;

— `\t` : HT (tabulation horizontale) ;

— `\\` : pour afficher `\`.

2 Normes ISO 8859

2.1 Normes ISO 8859

Face à la diversité des caractères textuels existant, le format ASCII est insuffisant. En effet, il est uniquement adapté aux caractères utilisés en langue anglaise (pas d'accents, pas d'autres alphabets). Aussi, d'autres formats ont été développés à sa suite dans l'optique de maintenir une certain compatibilité avec l'ASCII. Pour cela, c'est la norme ISO 8859, développée par l'ISO (*International Organization for Standardization*), qui a d'abord été utilisée.

Cette norme décrit 16 **pages** différentes (dont 10 uniquement pour les langues latines) encodant les caractères sur 8 bits, en maintenant les 128 premiers codes de la table ASCII et en ajoutant 128 nouveaux caractères par page.

Ces pages peuvent être désignées sous la forme ISO-8859-n où $n \in \{1, ...16\}$ (ISO-8859-1, ISO8859-2, etc.), mais certaines pages sont parfois appelées par un nom d'usage particulier (latin-1, latin-2, etc.).

Code ISO (nom usuel)	**Zone linguistique**
8859-1 (latin-1)	Europe occidentale
8859-2 (latin-2)	Europe centrale et de l'est
8859-3 (latin-3)	Europe du sud
8859-4 (latin-4)	Europe du nord
8859-5	Cyrilique
8859-6	Arabe
8859-7	Grec
8859-8	Hébreu
8859-9 (latin-5)	Turc, Kurde
8859-10 (latin-6)	Nordique (réarrangement du latin-4)
8859-11	Thaï
8859-12	Devanagari (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

FIGURE 1 – Pages de la norme ISO-8859

3 Codage Unicode

3.1 UCS, points de code

La norme ISO-8859 permet l'encodage d'un grand nombre de caractères. Cependant, elle ne permet pas l'encodage dans une même page de caractères provenant de zones linguistiques différentes. Pour palier à ce problème l'ISO a développé un jeu universel de caractère appelé UCS (*Universal Character Set*) sous la norme appelée ISO-10646. Cette norme associe à chaque caractère pris en compte un unique nom (en anglais et en français) et un numéro (entier positif) appelé **point de code**.

Aujourd'hui, plus de 110 000 caractères ont été recensés avec comme objectif de recenser tous les caractères utilisés dans n'importe quelle langue, pour une capacité maximale de recensement de 4 294 967 295 caractères (32 bits utilisés).

Par soucis de compatibilité, les 256 premiers points de code sont ceux de la norme ISO-8859-1 (latin 1) et donc les 127 premiers points de code sont ceux de l'ASCII.
On note les points de code `U+xxxx` pour désigner les points de code de l'UCS où chaque `x` représente un chiffre hexadécimal (on donne toujours au moins 4 chiffres et on en ajoute si nécessaire).

Par exemple, le caractère `'a'` a pour point de code $97 = (61)_{16}$ que l'on désigne donc par `U+0061`.

Autre exemple, le caractère `è` a pour point de code $232 = (e8)_{16}$ que l'on désigne donc par `U+00E8`.

Pour encoder tous les points de code de UCS en binaire, on aurait besoin a priori de 32 bits soit 4 octets par caractère d'un texte : on se rend compte que cela représenterait un grand gâchis de mémoire. Aussi, a été développée une norme appelée Unicode ou UTF (*Universal Transformation Format*), norme déclinée en trois versions : UTF-8, UTF-16 et UTF-32.

3.2 Intérêt et spécificités des encodages

Le format UTF-32 correspond à l'encodage naïf des points de code en binaire. Il présente le désavantage majeur de toujours utiliser 32 bits (4 octets) pour représenter chaque caractère. Aussi, un texte encodé en UTF-32 prendra beaucoup d'espace mémoire. En revanche, il facilite les algorithme de manipulation de texte (accès au n-ème caractère d'une chaîne, extraction de sous-chaîne, etc.) puisque chaque caractère est encodé sur le même nombre de bits.

Les formats UTF-8 et UTF-16 encodent les caractères sur un nombre variable de bits. Par exemple, en UTF-8, le caractère `'a'` de point de code `U+0061` est encodé sur 8 bits par le mot binaire `01100001` alors que le caractère de point de code `U+0A9C` est encodé sur 24 bits par `11100000 10101010 10011100`.

À noter : seul l'UTF-32 est capable de d'encoder l'intégralité des points de codes prévus par l'UCS. L'UTF-16 et l'UTF-8 n'encodant ces points de code que de `U+0000` à `U+10FFFF`.

Dans UTF-8 et UTF-16, le 8 et le 16 désignent le nombre **minimum** de bits nécessaire à l'encodage d'un caractère. Plus précisément, l'UTF-8 utilise entre 1 et 4 octets par caractère alors qu'UTF 16 en utilise toujours 2 ou 4.

Ces différences induisent des utilisations différentes pour les deux encodages :

— l'UTF-8 utilise en général moins d'espace mémoire que l'UTF-16, surtout dans les langues occidentales ;

— l'UTF-8 est en général plus difficile à traiter que l'UTF-16 car plus complexe.

Ainsi selon les nécessités des différentes situations, on observe une prévalence de l'un ou de l'autre. Par exemple, en réseau où il est important de limiter la taille des informations transmises pour économiser de la bande passante, on utilise plutôt l'UTF-8. Si en revanche la taille des données est moins importante que le temps de traitement, on utilise plutôt l'UTF-16 (c'est notamment le format utilisé par *Windows*).

Résumons tout cela :

Encodage	**Nombre d'octets**	**Traitement**	**Mémoire**
UTF-32	4	Très simple	Très gourmand
UTF-16	2 ou 4	Simple	Gourmand
UFT-8	1 à 4	Difficile	Économique

FIGURE 2 – Spécificités des encodages UTF

Encodage	**Contexte d'utilisation**
UTF-32	Algorithmes spécifiques de traitement de texte
UTF-16	Efficacité temporelle, exécution rapide importante
UFT-8	Efficacité mémoire, temps d'exécution peu important

FIGURE 3 – Utilisations des encodages UTF

3.3 Fonctionnement de l'UTF-8

Pour utiliser un nombre variable d'octets en UTF-8, on procède comme suit :

— Si le bit de poids fort est un `0`, un seul octet est utilisé et l'encodage est le même que l'encodage ASCII (hormis le bit de parité qui disparaît ici).

— Si le bit de poids fort est un `1`, alors plusieurs octets sont utilisés. Dans ce cas, le début du premier octet indique le nombre d'octets utilisés par une succession de `1` suivie d'un `0` et tous les autres octets commencent par `10`. Ces `0` et ces `1` ne sont présents que pour indiquer la forme de l'encodage et ne sont donc pas à prendre en compte dans la représentation du point de code représenté.

Cela donne, en écrivant `x` pour les bits codant le point de code :

1. pour 1 octet : `0xxxxxx`
2. pour 2 octets : `110xxxxx 10xxxxxx`
3. pour 3 octets : `1110xxxx 10xxxxxx 10xxxxxx`
4. pour 4 octets : `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

En particulier, les `x` représentant les bits disponibles pour écrire en binaire le point de code, on peut en déduire les plages des nombres représentables en fonction du nombre d'octet de la représentation :

Octets	Bits codant	Plage
1	7	U+0000 à U+007F
2	11	U+0080 à U+07FF
3	16	U+0800 à U+FFFF
4	21	U+10000 à U+10FFFF

FIGURE 4 – Plages de points de code en UTF-8

Exemple 1 : On veut encoder le caractère de point de code `U+01A3`. Étant donné la plage ci-dessus (ou simplement en comptant que `1A3` nécessite $1+4+4=9$ bits pour être représenté en binaire) on voit que 2 octets sont nécessaire pour le représenter en UTF-8. Il sera donc sous la forme :

110xxxxx 10xxxxxx

De plus on a $(\texttt{1A3})_{16} = (\texttt{001 1010 0011})_2$ (on a complété avec des zéros pour avoir 11 bits). La représentation en UTF-8 du caractère de point de code `U+01A3` est donc :

11000110 10100011

Exemple 2 : On veut décoder le caractère encodé en UTF-8 par :

11100111 10100011 10011010

Distinguons les bits de structure et les bits codant dans l'écriture :

11100111 10100011 10011010

Les bits de structure nous indiquent que 3 octets sont utilisés (ici cela n'est pas nécessaire mais lorsqu'on lit une succession d'octets représentant une suite de caractères cela est essentiel). Il suffit maintenant de traduire les 16 bits codant en hexadécimal pour obtenir le point de code du caractère encodé :

$$(\texttt{01111000 11011010})_2 = (\texttt{78DA})_{16}$$

On en déduit que le caractère encodé ici est celui de point de de code `U+78DA` soit 磚.

3.4 UTF-8 en Python

Les chaînes de caractères Python sont des séquences de caractères au format UTF-8. Quelques fonctionnalités Python peuvent s'avérer utiles :

— Si `s` est une chaîne de caractères, `s[i]` permet d'accéder au `i`-ème caractère de `s` (on commence à 0).

Exemple : si `s = 'coucou'`, alors `s[3] = 'c'`.

— Toujours pour `s` une chaîne de caractères, `len(s)` renvoie sa longueur.

Exemple : si `s = 'coucou'`, alors `len(s) = 3`

— Dans une chaîne de caractères, on peut insérer un symbole par le raccourci `\uhhhh` où `hhhh` désigne le point de code du caractère (en hexadécimal !).

Exemple : `'\u063\u06f\u075\u063\u06f\u075'` est équivalent à `'coucou'`.

— La commande `ord(c)` vue pour l'ASCII renvoie en fait le point de code de l'UCS (qui correspond au code ASCII pour les 128 premiers caractère).

Exemple : `ord(磚)` renvoie $30938 = (\texttt{78DA})_{16}$.

— La commande `chr(n)` vue pour l'ASCII, fonctionne également lorsque `n` est un point de code UCS et renvoie le caractère associé.

Exemple : `chr(30938)` renvoie le symbole 磚.

— Enfin, on pourra aussi se servir des fonctions :

1. `int(s, b)` (déjà vue dans les exercices du chapitre P1) convertissant la chaîne de caractères `s` en un entier selon une interprétation en base `b` ;
2. `hex(n)` renvoyant l'écriture hexadécimale de l'entier `n` ;
3. `bin(n)` renvoyant l'écriture bianaire de l'entier `n` ;