Python – Représentation des textes

Codage ASCII

Python utilise un encodage des caractères compatible avec le code ASCII pour les caractères de base. Il est possible d'obtenir le numéro associé à chaque caractère ou au contraire d'obtenir le caractère à partir du numéro.

```
>>> ord("a")
97
>>> chr(97)
'a'
```

Les fonctions suivantes permettent de coder et de décoder des textes en ASCII :

```
def encodage(texte):
    return [ord(c) for c in texte]

def decodage(liste):
    return "".join([chr(n) for n in liste])
```

Exercice 1 : Décoder le message suivant :

```
[66, 114, 97, 118, 111, 44, 32, 116, 117, 32, 97, 115, 32, 114, 101, 117,
115, 115, 105, 32, 108, 39, 101, 120, 101, 114, 99, 105, 99, 101, 32, 49]
```

```
>>> hex(ord("b"))
'0x62'
>>> chr(0x62)
'b'
```

La notation `0x` suivie d'un nombre en hexadécimal permet de donner une valeur en hexadécimale en Python.
On peut aussi indiquer quel caractère utiliser dans un texte en utilisant son code hexadécimal avec la notation `\xHH` où `HH` sont deux chiffres en hexadécimal.

```
>>> "J\x27ai\x20faim\x2E"
"J'ai faim."
>>> len("\x62")
1
```

On peut remarquer que même si `"\x62"` semble être composé de 4 caractères, il n'y en a en fait qu'un seul. C'est parce que \ est un **caractère d'échappement**. Cela signifie qu'il est le début d'une commande spéciale. Dans ce cas, c'est un moyen de définir un symbole par son code en hexadécimal. Il existe plusieurs caractères spéciaux, mais les principaux sont donnés ci-contre.

\a	BEL	son sur le haut parleur
\b	BS	suppression
\r	CR	retour chariot
\n	LF	nouvelle ligne
\t	HT	tabulation horizontale
\\		pour afficher \
\'		pour afficher '
\"		pour afficher "

```
>>> print("Houla c'\best bizarre\r ce\n qui\t m'arrive\a")
 cela cest bizarre
 qui     m'arrive
```

Le \b fait revenir d'un caractère à gauche la suite du texte, ce qui efface le ', le \r fait revenir au début de la ligne et écrase donc le début de la phrase. Le \n fait aller à la ligne, le \t rajoute une tabulation et enfin le \a produit un son sur le haut parleur de l'ordinateur.

ISO-8859-1, UTF-8 ou autre

Python n'est pas limité aux caractères ASCII.

```
>>> [ord(e) for e in "éèàêï"]
[233, 232, 224, 234, 239]
>>> ''.join([chr(c) for c in [192, 198, 181, 182, 215]])
'ÀÆµ¶×'
```

Les codes obtenus sont ceux de la table ISO-8859-1, ou latin-1. Pour des raisons de place en mémoire va prendre la représentation la plus "petite" pour chaque caractère : ASCII ou latin-1 pour les premiers caractères, puis UTF-8 pour les autres. Les caractères peuvent également être donnés directement à partir de leur code unicode :

```
>>> "\u0043\u0069\u0065\u006C\u0020\u00c9\u0074\u006F\u0069\u006C\u00E9"
'Ciel Étoilé'
>>> print("\U0001f631 \u0065\u006d\u006f\u006a\u0069\u0073 \U0001f601 !!!")
😱 emojis 😁 !!!
```

Il est même possible de donner la description du caractère pour l'obtenir :

```
>>> print("\N{Latin Small Letter E with Acute}\N{SMILING FACE WITH HORNS}")
é😈
```

Python peut convertir n'importe quel texte dans n'importe quel encodage, à condition que les caractères utilisés existent dans ce encodage.

```
>>> "é".encode("utf8")
b'\xc3\xa9'
>>> "é".encode("latin1")
b'\xe9'
>>> "é".encode()
b'\xc3\xa9'
```

Les objets que l'on obtient ne sont pas des textes, mais des suites d'octets, identifiées par le b avant les apostrophes. On remarque que l'encodage de base proposé par Python est l'UTF-8, qui garantit que tout peut être encodé. On remarque également qu'en latin-1, le caractère n'est codé que sur un octet, contre 2 pour l'UTF-8. C'est pourquoi en interne, Python utilise le latin-1 lorsque c'est possible, pour économiser de l'espace mémoire. Voici les versions en binaire de ces octets.

```
>>> ' '.join([f"{c:08b}" for c in 'é'.encode('utf8')])
'11000011 10101001'
>>> ' '.join([f"{c:08b}" for c in 'é'.encode('latin1')])
'11101001'
```

En UTF-8, le caractère "é" correspondant à U+00E9 en unicode, son écriture binaire est bien de la forme 110xxxxx 10xxxxxx.

À l'inverse, il est possible de décoder une suite d'octets, et c'est là que les choses peuvent se compliquer.

```
>>> b'\xc3\xa9'.decode("utf8")
'é'
>>> b'\xc3\xa9'.decode("latin1")
'Ã©'
>>> b'\xe9'.decode("latin1")
'é'
>>> b'\xe9'.decode("utf8")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 0: unexp
ected end of data
```

Lorsqu'on decode une suite d'octets avec le bon encodage, on retrouve le texte voulu. Par contre, si on utilise un autre encodage, on obtient soit des caractères qui ne sont pas les bons, soit une erreur parce qu'une suite d'octets n'existe pas dans l'encodage choisi. Ainsi, les octets C3 et A9 correspondent à deux symboles différents en latin-1. De même, l'octet E9, correspond au début d'une suite de 3 octets de la forme 1110xxxx 10xxxxxx 10xxxxxx.

Exercice 2 : En sachant que $10000000_2 = 128_{10}$ et $10111111_2 = 191_{10}$, trouver deux nombres `n1` et `n2` entre 128 et 191 tels que `"é"+chr(n1)+chr(n2)` donne une chaîne de 3 caractères existants (pas de `\xHH`) et que `("é"+chr(n1)+chr(n2)).encode("latin1").decode("utf8")` donne également un symbole valide.

Convertir un fichier

Il se peut qu'à l'ouverture d'un fichier, les caractères ne s'affichent pas tous de façon correcte. C'est parce que l'encodage du fichier n'est pas reconnu ou n'est pas le bon. Pour régler cela, nous allons faire une fonction permettant de changer l'encodage d'un fichier. La fonction `open(fichier, action)` permet d'ouvrir un fichier pour le lire ou y écrire dedans. L'action peut être soit `"r"` pour lire, soit `"w"` pour écrire. Il est également possible de rajouter un paramètre pour préciser l'encodage. Normalement, il faut fermer le fichier à la fin de son utilisation, mais Python propose une syntaxe spéciale qui permet de manipuler le fichier puis de le fermer automatiquement.
On considère la fonction suivante :

```
def creation_fichier_utf8():
    with open("fichier_en_utf8.txt", "w", encoding="utf8") as fichier:
        fichier.write("Ce fichier est enregistré en utf8.\n")
        fichier.write("Si tout se passe bien, il ne devrait pas y avoir")
        fichier.write(" de caractères bizarres.\n")
        fichier.write("Même si on fait : éÉèÉàÀùÙ.\n")
```

Cette fonction permet de créer un fichier en UTF-8 dans le même dossier que votre fichier Python.

Exercice 3 : Écrire une fonction `creation_fichier_latin1()` qui fait la même chose mais avec un encodage `"latin1"` et crée un fichier `fichier_en_latin1.txt`.
On va maintenant faire une fonction permettant d'ouvrir un fichier avec un encodage donné et qui va l'afficher en intégralité.

```
def lire_fichier(nom, encodage="utf8"):
    with open(nom, "r", encoding = encodage) as fichier:
        print(fichier.read())
```

Par défaut, l'encodage sera l'UTF-8 si aucun encodage n'est donné.

Exercice 4 : Ouvrir vos deux fichiers avec les bons encodages, ainsi qu'avec le mauvais.

On peut remarquer que les fichiers sont ouverts d'un seul coup. Il est également possible de les ouvrir ligne par ligne.

```
def lire_fichier_par_ligne(nom, encodage="UTF-8"):
    with open(nom, "r", encoding = encodage) as fichier:
        for ligne in fichier.readlines():
            print(ligne)
```

Si vous testez votre fonction, vous remarquerez qu'une ligne est ajoutée entre chaque ligne. C'est parce que le dernier caractère de chaque ligne du fichier est un retour à la ligne et qu'il rajoute donc une ligne en plus à l'affichage. Pour l'enlever, il suffit de mettre `print(ligne[:-1])`.

Exercice 5 : Écrire une fonction `points_de_code_fichier(nom, encodage)` qui affiche ligne par ligne la liste des points de code du fichier ouvert avec l'encodage donné. Comparer les résultats obtenus avec `fichier_en_utf8.txt` en UTF-8 et en latin-1 ainsi que `fichier_en_latin1.txt` en latin-1.

Exercice 6 : Écrire une fonction `conversion_fichier(nom1, nom2, encodage1, encodage2)` qui ouvre un fichier `nom1` avec l'encodage `encodage1` et le copie dans un fichier `nom2` avec l'encodage `encodage1`. Il faut mieux ouvrir le premier fichier, enregistrer le résultat de `fichier1.read()` puis ouvrir le deuxième fichier et y écrire le contenu obtenu.

Et les autres encodages

Python connait bien sûr bien plus d'encodages que ASCII, latin-1 et UTF-8. Il est possible d'obtenir la liste de ces encodages en faisant :

```
>>> import encodings
>>> print(sorted(set(encodings.aliases.aliases.values())))
```

On obtient ainsi la liste triée des encodages reconnus, en supprimant les doublons (comme `"uft8"` et `"utf-8"`).

Exercice 7 : Trouver le nombre d'encodages disponibles.

Pour aller plus loin

Il est possible d'afficher les descriptions des différents caractères unicodes. Pour cela, il faut d'abord rajouter `from unicodedata import *` au début de votre script Python et rajouter la fonction suivante :

```
def afficher_information(debut, fin):
    for i in range(debut, fin):
        c = chr(i)
        if name(c,'-') != '-':
            print(f"{c} | {ord(c):04x} | {name(c,'-')}")
```

Exercice 8 : Tester cette fonction pour différentes plages de caractères unicode. Vous pouvez par exemple chercher le code correspondant à un emoji et chercher les symboles avec des codes proches.