Python et le casse-tête des caractères accentués dans les textes

Un jour ou l'autre, le développeur sous Python sera confronté à un message d'erreur dû à un caractère accentué. Exemple :
`UnicodeEncodeError: 'ascii' codec can't encode characters …`

Pourquoi ? Parce que sous Python la gestion des textes (chaînes de caractères) n'est vraiment pas simple...
Dans la plupart des langages de développement, on peut travailler avec un seul type de chaînes : "String". Mais Python (jusqu'à la version 2.7.x) nous oblige à jongler avec 2 types : **unicode** et **str** (à partir de la version 3.0 de Python, il n'y a plus de str). Certaines fonctions retournent des textes de type unicode, d'autres des str. Certaines réclament des entrées en unicode, d'autres en str.

Essayons déjà d'expliquer la différence entre les deux types.

Le type unicode :

Il attribue à chaque caractère un code unique. Mais surtout, il couvre plusieurs dizaines de langues avec tous leurs symboles ou glyphes. C'est un codage quasi universel des lettres ou syllabes, chiffres ou nombres, symboles divers, signes diacritiques et signes de ponctuation. D'où son nom : UNICODE. Pour créer un texte unicode sous Python, il faut l'écrire entre guillemets précédés de la lettre u : **u**"ceci est une chaine unicode" ou bien **u**'Encore un texte unicode'.

Le type str :

Il code chaque caractère sur un seul octet ce qui n'offre que 256 possibilités (mais il y a des exceptions : voir le § sur l'UTF-8 sur cette page). Cela ne permet donc pas de décrire les dizaines de milliers de glyphes qui existent de par le monde. C'est pourquoi il existe des jeux de caractères ("charset" en anglais ; Python les appelle "**codec**" ou "**encoding**") qui sont des tables qui listent les caractères et symboles d'un alphabet ou système d'écriture.
Un nombre (entre 0 et 255) = un symbole particulier dans un codec donné.
Exemple : dans le codec "latin1" le caractère **à** a la valeur 224.

Le codec par défaut des str dans Python est souvent l'ASCII. Pour vérifier :

```
>>> import sys ; sys.getdefaultencoding()
```

Hélas, c'est une norme américaine qui a "oublié" les caractères accentués (et qui refuse les symboles dont le code est supérieur à 127) ! **La présence d'un simple accent dans une str peut provoquer le plantage d'une fonction...**
Nous verrons plus loin les moyens de maîtriser l'usage des str. Mais en général, utiliser une variable de type **str** impose de garder à l'esprit que ses symboles sont associés à un codec particulier.

Maîtriser les échanges entre unicode et str :

Transformer une unicode en str, c'est **l'encodage** vers un codec particulier :

```
>>> texteStr = texteUnicode.encode("utf-8")
```

Transformer une str en unicode, c'est **le décodage** depuis un codec :

```
>>> texteUnicode = texteStr.decode("latin1")
```

Comme le codec par défaut des str a un problème avec les accents, on est obligé d'utiliser le type unicode pour les textes qui en contiennent. Exemple:

```
>>> saison = u"été"
```

Heureusement, la plupart des fonctions internes de Python acceptent indifféremment des textes unicode ou str. Elles se chargent elles-mêmes du décodage ou de l'encodage selon leurs exigences propres.
Malheureusement, si elles ont besoin de décoder une str en unicode (ou d'encoder une unicode en str), **elles vont systématiquement utiliser le codec ASCII et cela provoque une erreur Python quand le texte contient des caractères non-ascii comme : à, æ, ç, è, é, ê, ë, ï, Æ, Ç, È**, etc. (voici la liste très limitée des 128 caractères ASCII)

Autre source d'erreurs : les échanges avec l'extérieur du programme Python :

- l'accès aux dossiers et fichiers par leur nom qui peut contenir des accents,
- la lecture ou l'écriture du contenu des fichiers (un fichier peut être encodé dans n'importe lequel des codecs !),
- la saisie des utilisateurs et l'affichage des textes à l'écran,
- les échanges de données par le réseau

Des solutions !

Concaténer unicode + str :

Cela donne toujours une unicode. Et ça commence par décoder la str avec le codec ASCII.
Donc si la str contient des accents : Erreur garantie !
Solution : convertir soi-même la str en unicode (avec le bon codec : bon courage !)

```
>>> chemin = monDossierStr.decode('cp1252') + u"/étape1.txt"
```

voir le § suivant pour le codec des noms de fichiers

Le nom des fichiers et dossiers :

La plupart des fonctions qui utilisent des chemins d'accès et des noms de fichiers gèrent très bien l'unicode.

```
>>> f = open( u"D:\\Février\\rapport.txt" )  # si le fichier existe, ça marche
>>> f = open( "D:\\Février\\rapport.txt" )    # échec : il ne trouvera pas le fichier !
>>> os.rename( "D:\\Février\\rapport1.txt", "D:\\Février\\rapportFinal.txt" )  # Erreur : chemin introuvable
```

Par contre, les fonctions Python retournent des chemins vers des noms de fichiers sous forme de str (il y a des exceptions).

```
>>> os.getcwd()  # retourne une str
>>> cheminPlugin = os.path.abspath(os.path.dirname(__file__))  # c'est une str
```

Pour trouver le codec utilisé par le système de fichier sur lequel on exécute le code :

```
>>> import locale
>>> codecNomsFichiers = locale.getpreferredencoding()
```

Donc la ligne suivante va fonctionner si le fichier existe (ouvre le fichier étape1.txt dans le répertoire de travail) :

```
>>> f = open( os.getcwd().decode(codecNomsFichiers) + u"/étape1.txt" )
```

Mais certaines fonctions ou certains modules comme **subprocess** plantent quand on leur passe des chemins avec accents en unicode. Il faut donc les encoder avec locale.getpreferredencoding()

```
>>> subprocess.call( u"D:\\Réseau\\Supervision.exe".encode(codecNomsFichiers) )  # et oui, c'est ch**** !
```

Lire ou écrire le contenu des fichiers

Il y a toutes sortes de fonctions et de modules qui lisent ou écrivent des fichiers. Mais il est difficile de trouver comment elles gèrent les textes : unicode ou str, quel codec…

Concentrons-nous sur l'objet Fichier de Python (file object), retourné par la fonction open() :

```
>>> f = open("mon/fichier")
>>> txt = f.read()  # txt est une str encodée comme le fichier lu
```

Les problèmes commencent :

- Il n'y a aucun moyen simple et fiable de deviner l'encodage d'un fichier par son simple contenu (certains types de fichiers comme le XML proposent des normes pour déclarer leur encodage ; mais il faut des fonctions spéciales pour les lire)
- Erreur garantie si on tente d'écrire (write) une unicode avec accent dans un fichier ouvert par la fonction open de base.

Maîtriser l'encodage de ses propres fichiers

En commançant par l'encodage de ses scripts python :

Il faut connaître ou même choisir comment ces fichiers sont enregistrés. Sous Windows, c'est l'encodage **mbcs** (Windows l'appelle ANSI) par défaut. Sous Linux, c'est souvent **utf-8** par défaut.

Ensuite, placer la ligne suivante au début du code pour indiquer un encodage utf-8 :

```
# -*- coding: utf-8 -*-
```

ou bien en ANSI (mbcs) :

```
# -*- coding: mbcs -*-
```

Assurer le travail sur les données :

1. Décoder tout de suite en unicode les textes lus depuis un fichier
2. Travailler avec des chaînes unicode
3. Les encoder au moment de les écrire dans un fichier

Première solution : open > lire > decode et encode > écrire

```
>>> f = open("mon/fichier")
>>> txt = f.read().decode('mbcs')  # txt est une unicode décodée depuis un fichier en ANSI
>>> fic = open("un/autre/fichier")
>>> fic.write( montxtUnicode.encode('mbcs') )  # écrire en ANSI dans ce fichier
```

Deuxième solution : utiliser le module **codecs** et sa fonction open pour manipuler directement des chaînes unicodes :

```
>>> import codecs
>>> f = codecs.open("mon/fichier", encoding='utf-8')  # on précise l'encodage à l'ouverture
>>> txt = f.read()  # txt est une unicode
…
>>> f = codecs.open("c:\\texte.txt", "w", encoding='mbcs')  # on force l'écriture en ANSI
>>> f.write( u"éèçЄù" )  # ça marche !
>>> f.write( "éèçЄù" )    # Erreur : il ne faut pas lui passer des str avec accents !
```

Remarque : le codec par défaut pour le contenu des fichiers n'est pas forcément le même que le codec pour le nom des fichiers. Facile à trouver :

```
>>> import sys
>>> sys.getfilesystemencoding()  # sous windows c'est: 'mbcs'
```

L'encodage des fichiers d'une autre source

C'est vraiment problématique car cela dépend du système, de l'application qui les a générés ou des paramètres de cette application.
Par exemple, l'éditeur de texte Notepad++ propose des dizaines d'encodages différents. Mais il est très doué pour deviner l'encodage des fichiers existants : ça peut rendre service quand on doit traiter un fichier externe qui contient des accents…

Se tromper d'encodage entre mbcs et utf8 provoquera peu de blocages. Mais l'interprétation des caractères accentués sera très moche. Par exemple :

```
>>> print u'éèçàù'.encode('utf8').decode('mbcs')
Ã©Ã¨Ã§Ã Ã¹
```

Si on veut éviter ces hiéroglyphes, les fonctions encode et decode proposent une autre option : une gestion moins stricte des erreurs de codage. On peut ainsi utiliser le codage ascii et lui demander d'ignorer les caractères qu'il ne connaît pas :

```
>>> txt = u"Le problème est résolu"
>>> txt.encode( 'ascii', errors='ignore' )  # élimine tout simplement les caractères accentués
'Le problme est rsolu'
```

Enfin, on peut obtenir un texte potable en remplaçant les caractères spéciaux par leur équivalent ASCII le plus proche. Avec l'alphabet latin, c'est très facile : [1]

```
>>> import unicodedata
>>> unicodedata.normalize('NFKD', u"ñéèçàîûö").encode('ascii', 'ignore')  # une str sans accent
'neecaiuo'
```

[1] Voici des explications très intéressantes :
http://sametmax.com/lencoding-en-python-une-bonne-fois-pour-toute/

Quelques mots sur l'usage de l'interface graphique Qt

Il faut passer des unicodes aux fonctions Qt. Voici quelques tutoriels sur Qt
Jusqu'à Qgis 1.8, Qt renvoit des textes en QString qu'on transforme en unicode très simplement avec la fonction unicode(laQStringRetourneeParQt).
Depuis Qgis 2.0 il n'y a plus de QString : que des unicodes.