Guide Unicode

Version 3.6.11

Guido van Rossum
and the Python development team

juin 29, 2020

Python Software Foundation
Email : docs@python.org

Table des matières

Version 1.12

This HOWTO discusses Python support for Unicode, and explains various problems that people commonly encounter when trying to work with Unicode.

1 Introduction à Unicode

1.1 Histoire des codes de caractères

En 1968, l'*American Standard Code for Information Interchange*, mieux connu sous son acronyme *ASCII*, a été normalisé. L'ASCII définissait des codes numériques pour différents caractères, les valeurs numériques s'étendant de 0 à 127. Par exemple, la lettre minuscule « a » est assignée à 97 comme valeur de code.

ASCII était une norme développée par les États-Unis, elle ne définissait donc que des caractères non accentués. Il y avait « e », mais pas « é » ou « Í ». Cela signifiait que les langues qui nécessitaient des caractères accentués ne pouvaient pas être fidèlement représentées en ASCII. (En fait, les accents manquants importaient pour l'anglais aussi, qui contient des mots tels que « naïve » et « café », et certaines publications ont des styles propres qui exigent des orthographes tels que « *coöperate* ».)

Pendant un certain temps, les gens ont juste écrit des programmes qui n'affichaient pas d'accents. Au milieu des années 1980, un programme Apple II BASIC écrit par un français pouvait avoir des lignes comme celles-ci :

```
PRINT "MISE A JOUR TERMINEE"
PRINT "PARAMETRES ENREGISTRES"
```

Ces messages devraient contenir des accents (terminée, paramètre, enregistrés) et ils ont juste l'air anormaux à quelqu'un lisant le français.

Dans les années 1980, presque tous les ordinateurs personnels étaient à 8 bits, ce qui signifie que les octets pouvaient contenir des valeurs allant de 0 à 255. Les codes ASCII allaient seulement jusqu'à 127, alors certaines machines ont assigné les valeurs entre 128 et 255 à des caractères accentués. Différentes machines avaient des codes différents, cependant, ce qui a conduit à des problèmes d'échange de fichiers. Finalement, divers ensembles de valeurs couramment utilisés pour la gamme 128–255 ont émergé. Certains étaient de véritables normes, définies par l'Organisation internationale de normalisation, et certaines étaient des conventions *de facto* qui ont été inventées par une entreprise ou une autre et qui ont fini par se répandre.

255 caractères, ça n'est pas beaucoup. Par exemple, vous ne pouvez pas contenir à la fois les caractères accentués utilisés en Europe occidentale et l'alphabet cyrillique utilisé pour le russe dans la gamme 128–255, car il y a plus de 128 de tous ces caractères.

Vous pouviez écrire les fichiers avec des codes différents (tous vos fichiers russes dans un système de codage appelé *KOI8*, tous vos fichiers français dans un système de codage différent appelé *Latin1*), mais que faire si vous souhaitiez écrire un document français citant du texte russe ? Dans les années 80, les gens ont commencé à vouloir résoudre ce problème, et les efforts de standardisation Unicode ont commencé.

Unicode a commencé par utiliser des caractères 16 bits au lieu de 8 bits. 16 bits signifie que vous avez 2^16 = 65 536 valeurs distinctes disponibles, ce qui permet de représenter de nombreux caractères différents à partir de nombreux alphabets différents. Un des objectifs initiaux était de faire en sorte que Unicode contienne les alphabets de chaque langue humaine. Il s'avère que même 16 bits ne suffisent pas pour atteindre cet objectif, et la spécification Unicode moderne utilise une gamme de codes plus étendue, allant de 0 à 1 114 111 (`0x10FFFF` en base 16).

Il existe une norme ISO connexe, ISO 10646. Unicode et ISO 10646 étaient à l'origine des efforts séparés, mais les spécifications ont été fusionnées avec la révision 1.1 d'Unicode.

(Cette discussion sur l'historique d'Unicode est extrêmement simplifiée. Les détails historiques précis ne sont pas nécessaires pour comprendre comment utiliser efficacement Unicode, mais si vous êtes curieux, consultez le site du consortium Unicode indiqué dans les références ou la page Wikipédia pour Unicode (page en anglais) pour plus d'informations.)

1.2 Définitions

A **character** is the smallest possible component of a text. “A”, “B”, “C”, etc., are all different characters. So are “È” and “Í”. Characters are abstractions, and vary depending on the language or context you’re talking about. For example, the symbol for ohms (Ω) is usually drawn much like the capital letter omega (Ω) in the Greek alphabet (they may even be the same in some fonts), but these are two different characters that have different meanings.

The Unicode standard describes how characters are represented by **code points**. A code point is an integer value, usually denoted in base 16. In the standard, a code point is written using the notation `U+12CA` to mean the character with value `0x12ca` (4,810 decimal). The Unicode standard contains a lot of tables listing characters and their corresponding code points :

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
```

Strictly, these definitions imply that it’s meaningless to say “this is character `U+12CA`”. `U+12CA` is a code point, which represents some particular character ; in this case, it represents the character “ETHIOPIC SYLLABLE WI”. In informal contexts, this distinction between code points and characters will sometimes be forgotten.

Un caractère est représenté sur un écran ou sur papier par un ensemble d’éléments graphiques appelé **glyphe**. Le glyphe d’un A majuscule, par exemple, est deux traits diagonaux et un trait horizontal, bien que les détails exacts dépendent de la police utilisée. La plupart du code Python n’a pas besoin de s’inquiéter des glyphes ; trouver le bon glyphe à afficher est généralement le travail d’une boîte à outils GUI ou du moteur de rendu des polices d’un terminal.

1.3 Encodages

To summarize the previous section : a Unicode string is a sequence of code points, which are numbers from 0 through `0x10FFFF` (1,114,111 decimal). This sequence needs to be represented as a set of bytes (meaning, values from 0 through 255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an **encoding**.

The first encoding you might think of is an array of 32-bit integers. In this representation, the string « Python » would look like this :

```
   P           y           t           h           o           n
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00 6e 00 00 00
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

Cette représentation est simple mais son utilisation pose un certain nombre de problèmes.

1. Elle n’est pas portable ; des processeurs différents ordonnent les octets différemment.
2. Elle gâche beaucoup d’espace. Dans la plupart des textes, la majorité des points de code sont inférieurs à 127, ou à 255, donc beaucoup d’espace est occupé par des octets `0x00`. La chaîne ci-dessus occupe 24 octets, à comparer aux 6 octets nécessaires pour une représentation en ASCII. L’utilisation supplémentaire de RAM n’a pas trop d’importance (les ordinateurs de bureau ont des gigaoctets de RAM et les chaînes ne sont généralement pas si grandes que ça), mais l’accroissement de notre utilisation du disque et de la bande passante réseau par un facteur de 4 est intolérable.
3. Elle n’est pas compatible avec les fonctions C existantes telles que `strlen()`, il faudrait donc utiliser une nouvelle famille de fonctions, celle des chaînes larges (*wide strings*).
4. De nombreuses normes Internet sont définies en termes de données textuelles et ne peuvent pas gérer le contenu incorporant des octets *zéro*.

Généralement, les gens n’utilisent pas cet encodage, mais optent pour d’autres encodages plus efficaces et pratiques. UTF-8 est probablement l’encodage le plus couramment pris en charge ; celui-ci sera abordé ci-dessous.

Les encodages n'ont pas à gérer tous les caractères Unicode possibles, et les plupart ne le font pas. Les règles pour convertir une chaîne Unicode en codage ASCII, par exemple, sont simples. pour chaque point de code :

1. Si le point de code est < 128, chaque octet est identique à la valeur du point de code.
2. Si le point de code est égal à 128 ou plus, la chaîne Unicode ne peut pas être représentée dans ce codage (Python déclenche une exception `UnicodeEncodeError` dans ce cas).

Latin-1, également connu sous le nom de ISO-8859-1, est un encodage similaire. Les points de code Unicode 0–255 étant identiques aux valeurs de Latin-1, la conversion en cet encodage nécessite simplement la conversion des points de code en octets de même valeur ; si un point de code supérieur à 255 est rencontré, la chaîne ne peut pas être codée en latin-1.

Les encodages ne doivent pas nécessairement être de simples mappages un à un, comme Latin-1. Prenons l'exemple du code EBCDIC d'IBM, utilisé sur les ordinateurs centraux IBM. Les valeurs de lettre ne faisaient pas partie d'un bloc : les lettres « a » à « i » étaient comprises entre 129 et 137, mais les lettres « j » à « r » étaient comprises entre 145 et 153. Si vous vouliez utiliser EBCDIC comme encodage, vous auriez probablement utilisé une sorte de table de correspondance pour effectuer la conversion, mais il s'agit en surtout d'un détail d'implémentation.

UTF-8 is one of the most commonly used encodings. UTF stands for « Unicode Transformation Format », and the "8" means that 8-bit numbers are used in the encoding. (There are also a UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) UTF-8 uses the following rules :

1. Si le point de code est < 128, il est représenté par la valeur de l'octet correspondant.
2. Si le point de code est ≥ 128, il est transformé en une séquence de deux, trois ou quatre octets, où chaque octet de la séquence est compris entre 128 et 255.

UTF-8 a plusieurs propriétés intéressantes :

1. Il peut gérer n'importe quel point de code Unicode.
2. A Unicode string is turned into a sequence of bytes containing no embedded zero bytes. This avoids byte-ordering issues, and means UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can't handle zero bytes.
3. Une chaîne de texte ASCII est également un texte UTF-8 valide.
4. UTF-8 est assez compact. La majorité des caractères couramment utilisés peuvent être représentés avec un ou deux octets.
5. Si des octets sont corrompus ou perdus, il est possible de déterminer le début du prochain point de code encodé en UTF-8 et de se resynchroniser. Il est également improbable que des données 8-bits aléatoires ressemblent à du UTF-8 valide.

1.4 Références

Le site du Consortium Unicode, en anglais, a des diagrammes de caractères, un glossaire et des versions PDF de la spécification Unicode. Préparez-vous à une lecture difficile. Une chronologie de l'origine et du développement de l'Unicode est également disponible sur le site.

To help understand the standard, Jukka Korpela has written an introductory guide to reading the Unicode character tables.

Another good introductory article was written by Joel Spolsky. If this introduction didn't make things clear to you, you should try reading this alternate article before continuing.

Les pages Wikipédia sont souvent utiles ; voir les pages pour « Codage des caractères » et UTF-8, par exemple.

2 Prise en charge Unicode de Python

Maintenant que vous avez appris les rudiments de l'Unicode, nous pouvons regarder les fonctionnalités Unicode de Python.

2.1 Le type *String*

Since Python 3.0, the language features a `str` type that contain Unicode characters, meaning any string created using `"unicode rocks!"`, `'unicode rocks!'`, or the triple-quoted string syntax is stored as Unicode.

L'encodage par défaut pour le code source Python est UTF-8, il est donc facile d'inclure des caractères Unicode dans une chaîne littérale :

```
try:
    with open('/tmp/input.txt', 'r') as f:
        ...
except OSError:
    # 'File not found' error message.
    print("Fichier non trouvé")
```

You can use a different encoding from UTF-8 by putting a specially-formatted comment as the first or second line of the source code :

```
# -*- coding: <encoding name> -*-
```

Note : Python 3 sait gérer les caractères Unicode dans les identifiants :

```
répertoire = "/tmp/records.log"
with open(répertoire, "w") as f:
    f.write("test\n")
```

Si vous ne pouvez pas entrer un caractère particulier dans votre éditeur ou si vous voulez garder le code source uniquement en ASCII pour une raison quelconque, vous pouvez également utiliser des séquences d'échappement dans les littéraux de chaîne (en fonction de votre système, il se peut que vous voyiez le glyphe réel du *delta majuscule* au lieu d'une séquence d'échappement `\u...`)

```
>>> "\N{GREEK CAPITAL LETTER DELTA}"  # Using the character name
'\u0394'
>>> "\u0394"                          # Using a 16-bit hex value
'\u0394'
>>> "\U00000394"                      # Using a 32-bit hex value
'\u0394'
```

De plus, une chaîne de caractères peut être créée en utilisant la méthode `decode()` de la classe `bytes`. Cette méthode prend un argument *encoding*, `UTF-8` par exemple, et optionnellement un argument *errors*.

L'argument *errors* détermine la réponse lorsque la chaîne en entrée ne peut pas être convertie selon les règles de l'encodage. Les valeurs autorisées pour cet argument sont `'strict'` (« strict » : lève une exception `UnicodeDecodeError`), `'replace'` (« remplacer » : utilise `U+FFFD, REPLACEMENT CARACTER`), `'ignore'` (« ignorer » : n'inclut pas le caractère dans le résultat Unicode) ou `'backslashreplace'` (« remplacer avec antislash » : insère une séquence d'échappement `\xNN`). Les exemples suivants illustrent les différences :

```
>>> b'\x80abc'.decode("utf-8", "strict")
Traceback (most recent call last):
    ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 0:
```

(suite sur la page suivante)

(suite de la page précédente)

```
  invalid start byte
>>> b'\x80abc'.decode("utf-8", "replace")
'\ufffdabc'
>>> b'\x80abc'.decode("utf-8", "backslashreplace")
'\\x80abc'
>>> b'\x80abc'.decode("utf-8", "ignore")
'abc'
```

Encodings are specified as strings containing the encoding's name. Python 3.2 comes with roughly 100 different encodings ; see the Python Library Reference at standard-encodings for a list. Some encodings have multiple names ; for example, `'latin-1'`, `'iso_8859_1'` and `'8859`" are all synonyms for the same encoding.

Des chaînes Unicode à un caractère peuvent également être créées avec la fonction native `chr()`, qui prend des entiers et renvoie une chaîne Unicode de longueur 1 qui contient le point de code correspondant. L'opération inverse est la fonction native `ord()` qui prend une chaîne Unicode d'un caractère et renvoie la valeur du point de code :

```
>>> chr(57344)
'\ue000'
>>> ord('\ue000')
57344
```

2.2 Conversion en octets

La méthode inverse de `bytes.decode()` est `str.encode()`, qui renvoie une représentation `bytes` de la chaîne Unicode, codée dans l'encodage *encoding* demandé.

Le paramètre *errors* est le même que le paramètre de la méthode `decode()` mais possède quelques gestionnaires supplémentaires. En plus de `'strict'`, `'ignore'` et `'remplace'` (qui dans ce cas insère un point d'interrogation au lieu du caractère non codable), il y a aussi `'xmlcharrefreplace'` (insère une référence XML), `backslashreplace` (insère une séquence `\uNNNN`) et `namereplace` (insère une séquence `\N{...}`).

L'exemple suivant montre les différents résultats :

```
>>> u = chr(40960) + 'abcd' + chr(1972)
>>> u.encode('utf-8')
b'\xea\x80\x80abcd\xde\xb4'
>>> u.encode('ascii')
Traceback (most recent call last):
    ...
UnicodeEncodeError: 'ascii' codec can't encode character '\ua000' in
  position 0: ordinal not in range(128)
>>> u.encode('ascii', 'ignore')
b'abcd'
>>> u.encode('ascii', 'replace')
b'?abcd?'
>>> u.encode('ascii', 'xmlcharrefreplace')
b'&#40960;abcd&#1972;'
>>> u.encode('ascii', 'backslashreplace')
b'\\ua000abcd\\u07b4'
>>> u.encode('ascii', 'namereplace')
b'\\N{YI SYLLABLE IT}abcd\\u07b4'
```

Les routines de bas niveau pour enregistrer et accéder aux encodages disponibles se trouvent dans le module `codecs`. L'implémentation de nouveaux encodages nécessite également de comprendre le module `codecs`. Cependant, les fonctions d'encodage et de décodage renvoyées par ce module sont généralement de bas-niveau pour être facilement utilisées et l'écriture de nouveaux encodages est une tâche très spécialisée, donc le module ne sera pas couvert dans ce guide.

2.3 Littéraux Unicode dans le code source Python

Dans le code source Python, des points de code Unicode spécifiques peuvent être écrits en utilisant la séquence d'échappement `\u`, suivie de quatre chiffres hexadécimaux donnant le point de code. La séquence d'échappement `\U` est similaire, mais attend huit chiffres hexadécimaux, pas quatre :

```
>>> s = "a\xac\u1234\u20ac\U00008000"
... #     ^^^^ two-digit hex escape
... #          ^^^^^^ four-digit Unicode escape
... #                 ^^^^^^^^^^ eight-digit Unicode escape
>>> [ord(c) for c in s]
[97, 172, 4660, 8364, 32768]
```

L'utilisation de séquences d'échappement pour des points de code supérieurs à 127 est acceptable à faible dose, mais devient gênante si vous utilisez beaucoup de caractères accentués, comme c'est le cas dans un programme avec des messages en français ou dans une autre langue utilisant des lettres accentuées. Vous pouvez également assembler des chaînes de caractères à l'aide de la fonction native `chr()`, mais c'est encore plus fastidieux.

Idéalement, vous devriez être capable d'écrire des littéraux dans l'encodage naturel de votre langue. Vous pourriez alors éditer le code source de Python avec votre éditeur favori qui affiche les caractères accentués naturellement, et a les bons caractères utilisés au moment de l'exécution.

Python considère que le code source est écrit en UTF-8 par défaut, mais vous pouvez utiliser presque n'importe quel encodage si vous déclarez l'encodage utilisé. Cela se fait en incluant un commentaire spécial sur la première ou la deuxième ligne du fichier source :

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

u = 'abcdé'
print(ord(u[-1]))
```

La syntaxe s'inspire de la notation d'*Emacs* pour spécifier les variables locales à un fichier. *Emacs* supporte de nombreuses variables différentes, mais Python ne gère que *coding*. Les symboles `-*-` indiquent à *Emacs* que le commentaire est spécial ; ils n'ont aucune signification pour Python mais sont une convention. Python cherche `coding: name` ou `coding=name` dans le commentaire.

Si vous n'incluez pas un tel commentaire, l'encodage par défaut est UTF-8 comme déjà mentionné. Voir aussi la **PEP 263** pour plus d'informations.

2.4 Propriétés Unicode

The Unicode specification includes a database of information about code points. For each defined code point, the information includes the character's name, its category, the numeric value if applicable (Unicode has characters representing the Roman numerals and fractions such as one-third and four-fifths). There are also properties related to the code point's use in bidirectional text and other display-related properties.

Le programme suivant affiche des informations sur plusieurs caractères et affiche la valeur numérique d'un caractère particulier :

```
import unicodedata

u = chr(233) + chr(0x0bf2) + chr(3972) + chr(6000) + chr(13231)

for i, c in enumerate(u):
    print(i, '%04x' % ord(c), unicodedata.category(c), end=" ")
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print(unicodedata.name(c))

# Get numeric value of second character
print(unicodedata.numeric(u[1]))
```

Si vous l'exécutez, cela affiche :

```
0 00e9 Ll LATIN SMALL LETTER E WITH ACUTE
1 0bf2 No TAMIL NUMBER ONE THOUSAND
2 0f84 Mn TIBETAN MARK HALANTA
3 1770 Lo TAGBANWA LETTER SA
4 33af So SQUARE RAD OVER S SQUARED
1000.0
```

Les codes de catégorie sont des abréviations décrivant la nature du caractère. Celles-ci sont regroupées en catégories telles que « Lettre », « Nombre », « Ponctuation » ou « Symbole », qui sont à leur tour divisées en sous-catégories. Pour prendre par exemple les codes de la sortie ci-dessus, `'Ll'` signifie « Lettre, minuscules », `'No'` signifie « Nombre, autre », `'Mn'` est « Marque, non-espaçant », et `'So'` est « Symbole, autre ». Voir la section Valeurs générales des catégories de la documentation de la base de données de caractères Unicode (ressource en anglais) pour une liste de codes de catégories.

2.5 Expressions régulières Unicode

Les expressions régulières gérées par le module `re` peuvent être fournies sous forme de chaîne d'octets ou de texte. Certaines séquences de caractères spéciaux telles que `\d` et `\w` ont des significations différentes selon que le motif est fourni en octets ou en texte. Par exemple, `\d` correspond aux caractères `[0-9]` en octets mais dans les chaînes de caractères correspond à tout caractère de la catégorie `'Nd'`.

Dans cet exemple, la chaîne contient le nombre 57 écrit en chiffres arabes et thaïlandais :

```
import re
p = re.compile(r'\d+')

s = "Over \u0e55\u0e57 57 flavours"
m = p.search(s)
print(repr(m.group()))
```

Une fois exécuté, `\d+` correspond aux chiffres thaïlandais et les affiche. Si vous fournissez le drapeau `re.ASCII` à `compile()`, `\d+` correspond cette fois à la chaîne « 57 ».

De même, `\w` correspond à une grande variété de caractères Unicode mais seulement `[a-zA-Z0-9_]` en octets (ou si `re.ASCII` est fourni) et `\s` correspond soit aux caractères blancs Unicode soit aux caractères `[ \t\n\r\f\v]`.

2.6 Références

Quelques bonnes discussions alternatives sur la gestion d'Unicode par Python sont :

- Processing Text Files in Python 3, par Nick Coghlan.
- Pragmatic Unicode, a PyCon 2012 presentation by Ned Batchelder.

Le type `str` est décrit dans la référence de la bibliothèque Python à textseq.

La documentation du module `unicodedata`.

La documentation du module `codecs`.

Marc-André Lemburg a donné une présentation intitulée « Python et Unicode » (diapositives PDF) à *EuroPython* 2002. Les diapositives sont un excellent aperçu de la conception des fonctionnalités Unicode de Python 2 (où le type de chaîne Unicode est appelé `unicode` et les littéraux commencent par `u`).

3 Lecture et écriture de données Unicode

Une fois que vous avez écrit du code qui fonctionne avec des données Unicode, le problème suivant concerne les entrées/sorties. Comment obtenir des chaînes Unicode dans votre programme et comment convertir les chaînes Unicode dans une forme appropriée pour le stockage ou la transmission ?

Il est possible que vous n'ayez rien à faire en fonction de vos sources d'entrée et des destinations de vos données de sortie ; il convient de vérifier si les bibliothèques utilisées dans votre application gèrent l'Unicode nativement. Par exemple, les analyseurs XML renvoient souvent des données Unicode. De nombreuses bases de données relationnelles prennent également en charge les colonnes encodées en Unicode et peuvent renvoyer des valeurs Unicode à partir d'une requête SQL.

Les données Unicode sont généralement converties en un encodage particulier avant d'être écrites sur le disque ou envoyées sur un connecteur réseau. Il est possible de faire tout le travail vous-même : ouvrir un fichier, lire un élément 8-bits, puis convertir les octets avec `bytes.decode(encoding)`. Cependant, l'approche manuelle n'est pas recommandée.

La nature multi-octets des encodages pose problème ; un caractère Unicode peut être représenté par plusieurs octets. Si vous voulez lire le fichier par morceaux de taille arbitraire (disons 1024 ou 4096 octets), vous devez écrire un code de gestion des erreurs pour détecter le cas où une partie seulement des octets codant un seul caractère Unicode est lue à la fin d'un morceau. Une solution serait de lire le fichier entier en mémoire et d'effectuer le décodage, mais cela vous empêche de travailler avec des fichiers extrêmement volumineux ; si vous avez besoin de lire un fichier de 2 Gio, vous avez besoin de 2 Gio de RAM (plus que ça, en fait, puisque pendant un moment, vous aurez besoin d'avoir à la fois la chaîne encodée et sa version Unicode en mémoire).

La solution serait d'utiliser l'interface de décodage de bas-niveau pour intercepter le cas des séquences d'encodage incomplètes. Ce travail d'implémentation a déjà été fait pour vous : la fonction native `open()` peut renvoyer un objet de type fichier qui suppose que le contenu du fichier est dans un encodage spécifié et accepte les paramètres Unicode pour des méthodes telles que `read()` et `write()`. Ceci fonctionne grâce aux paramètres *encoding* et *errors* de `open()` qui sont interprétés comme ceux de `str.encode()` et `bytes.decode()`.

Lire de l'Unicode à partir d'un fichier est donc simple :

```
with open('unicode.txt', encoding='utf-8') as f:
    for line in f:
        print(repr(line))
```

Il est également possible d'ouvrir des fichiers en mode « mise à jour », permettant à la fois la lecture et l'écriture :

```
with open('test', encoding='utf-8', mode='w+') as f:
    f.write('\u4500 blah blah blah\n')
    f.seek(0)
    print(repr(f.readline()[:1]))
```

Le caractère Unicode `U+FEFFF` est utilisé comme marque pour indiquer le boutisme (c'est-à-dire l'ordre dans lequel les octets sont placés pour indiquer une valeur sur plusieurs octets, *byte-order mark* en anglais ou *BOM*), et est souvent écrit en tête (premier caractère) d'un fichier afin d'aider à l'auto-détection du boutisme du fichier. Certains encodages, comme UTF-16, s'attendent à ce qu'une *BOM* soit présente au début d'un fichier ; lorsqu'un tel encodage est utilisé, la *BOM* sera automatiquement écrite comme premier caractère et sera silencieusement retirée lorsque le fichier sera lu. Il existe des variantes de ces encodages, comme `utf-16-le` et `utf-16-be` pour les encodages petit-boutiste et gros-boutiste, qui spécifient un ordre d'octets donné et ne sautent pas la *BOM*.

In some areas, it is also convention to use a « BOM » at the start of UTF-8 encoded files ; the name is misleading since UTF-8 is not byte-order dependent. The mark simply announces that the file is encoded in UTF-8. Use the “utf-8-sig” codec to automatically skip the mark if present for reading such files.

3.1 Noms de fichiers Unicode

Most of the operating systems in common use today support filenames that contain arbitrary Unicode characters. Usually this is implemented by converting the Unicode string into some encoding that varies depending on the system. For example, Mac OS X uses UTF-8 while Windows uses a configurable encoding ; on Windows, Python uses the name « mbcs » to refer to whatever the currently configured encoding is. On Unix systems, there will only be a filesystem encoding if you've set the `LANG` or `LC_CTYPE` environment variables ; if you haven't, the default encoding is UTF-8.

La fonction `sys.getfilesystemencoding()` renvoie l'encodage à utiliser sur votre système actuel, au cas où vous voudriez faire l'encodage manuellement, mais il n'y a pas vraiment de raisons de s'embêter avec ça. Lors de l'ouverture d'un fichier pour la lecture ou l'écriture, vous pouvez généralement simplement fournir la chaîne Unicode comme nom de fichier et elle est automatiquement convertie à l'encodage qui convient :

```
filename = 'filename\u4500abc'
with open(filename, 'w') as f:
    f.write('blah\n')
```

Les fonctions du module `os` telles que `os.stat()` acceptent également les noms de fichiers Unicode.

The `os.listdir()` function returns filenames and raises an issue : should it return the Unicode version of filenames, or should it return bytes containing the encoded versions ? `os.listdir()` will do both, depending on whether you provided the directory path as bytes or a Unicode string. If you pass a Unicode string as the path, filenames will be decoded using the filesystem's encoding and a list of Unicode strings will be returned, while passing a byte path will return the filenames as bytes. For example, assuming the default filesystem encoding is UTF-8, running the following program :

```
fn = 'filename\u4500abc'
f = open(fn, 'w')
f.close()

import os
print(os.listdir(b'.'))
print(os.listdir('.'))
```

produit la sortie suivante :

```
amk:~$ python t.py
[b'filename\xe4\x94\x80abc', ...]
['filename\u4500abc', ...]
```

La première liste contient les noms de fichiers encodés en UTF-8 et la seconde contient les versions Unicode.

Note that on most occasions, the Unicode APIs should be used. The bytes APIs should only be used on systems where undecodable file names can be present, i.e. Unix systems.

3.2 Conseils pour écrire des programmes compatibles Unicode

Cette section fournit quelques suggestions sur l'écriture de logiciels qui traitent de l'Unicode.

Le conseil le plus important est :

> Il convient que le logiciel ne traite que des chaînes Unicode en interne, décodant les données d'entrée dès que possible et encodant la sortie uniquement à la fin.

Si vous essayez d'écrire des fonctions de traitement qui acceptent à la fois les chaînes Unicode et les chaînes d'octets, les possibilités d'occurrences de bogues dans votre programme augmentent partout où vous combinez les deux différents types de chaînes. Il n'y a pas d'encodage ou de décodage automatique : si vous faites par exemple `str + octets`, une `TypeError` est levée.

Lors de l'utilisation de données provenant d'un navigateur Web ou d'une autre source non fiable, une technique courante consiste à vérifier la présence de caractères illégaux dans une chaîne de caractères avant de l'utiliser pour générer une ligne de commande ou de la stocker dans une base de données. Si vous le faites, vérifiez bien la chaîne décodée, pas les données d'octets codés ; certains encodages peuvent avoir des propriétés intéressantes, comme ne pas être bijectifs ou ne pas être entièrement compatibles avec l'ASCII. C'est particulièrement vrai si l'encodage est spécifié explicitement dans vos données d'entrée, car l'attaquant peut alors choisir un moyen intelligent de cacher du texte malveillant dans le flux de données encodé.

Conversion entre les encodages de fichiers

La classe `StreamRecoder` peut convertir de manière transparente entre les encodages : prenant un flux qui renvoie des données dans l'encodage #1, elle se comporte comme un flux qui renvoie des données dans l'encodage #2.

Par exemple, si vous avez un fichier d'entrée f qui est en Latin-1, vous pouvez l'encapsuler dans un `StreamRecoder` pour qu'il renvoie des octets encodés en UTF-8 :

```
new_f = codecs.StreamRecoder(f,
    # en/decoder: used by read() to encode its results and
    # by write() to decode its input.
    codecs.getencoder('utf-8'), codecs.getdecoder('utf-8'),

    # reader/writer: used to read and write to the stream.
    codecs.getreader('latin-1'), codecs.getwriter('latin-1') )
```

Fichiers dans un encodage inconnu

Vous avez besoin de modifier un fichier, mais vous ne connaissez pas son encodage ? Si vous savez que l'encodage est compatible ASCII et que vous voulez seulement examiner ou modifier les parties ASCII, vous pouvez ouvrir le fichier avec le gestionnaire d'erreurs `surrogateescape` :

```
with open(fname, 'r', encoding="ascii", errors="surrogateescape") as f:
    data = f.read()

# make changes to the string 'data'

with open(fname + '.new', 'w',
          encoding="ascii", errors="surrogateescape") as f:
    f.write(data)
```

The `surrogateescape` error handler will decode any non-ASCII bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the `surrogateescape` error handler is used when encoding the data and writing it back out.

3.3 Références

Une partie de la conférence Mastering Python 3 Input/Output (ressource en anglais), donnée lors de *PyCon* 2010 de David Beazley, parle du traitement de texte et du traitement des données binaires.

Le PDF du diaporama de la présentation de Marc-André Lemburg « Writing Unicodeaware Applications in Python » (ressource en anglais) traite des questions d'encodage de caractères ainsi que de l'internationalisation et de la localisation d'une application. Ces diapositives ne couvrent que Python 2.x.

The Guts of Unicode in Python (ressource en anglais) est une conférence *PyCon* 2013 donnée par Benjamin Peterson qui traite de la représentation interne Unicode en Python 3.3.

4 Remerciements

La première ébauche de ce document a été rédigée par Andrew Kuchling. Il a depuis été révisé par Alexander Belopolsky, Georg Brandl, Andrew Kuchling et Ezio Melotti.

Thanks to the following people who have noted errors or offered suggestions on this article : Éric Araujo, Nicholas Bastin, Nick Coghlan, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Terry J. Reedy, Chad Whitacre.

Index

P