
Guide Unicode

Version 3.13.0

Guido van Rossum and the Python development team

novembre 09, 2024

Python Software Foundation
Email : docs@python.org

Table des matières

1	Introduction à Unicode	1
1.1	Définitions	1
1.2	Encodages	2
1.3	Références	3
2	Prise en charge Unicode de Python	4
2.1	Le type <i>String</i>	4
2.2	Conversion en octets	5
2.3	Littéraux Unicode dans le code source Python	5
2.4	Propriétés Unicode	6
2.5	Comparaison de chaînes de caractères	7
2.6	Expressions régulières Unicode	8
2.7	Références	8
3	Lecture et écriture de données Unicode	8
3.1	Noms de fichiers Unicode	9
3.2	Conseils pour écrire des programmes compatibles Unicode	10
3.3	Références	11
4	Remerciements	11
	Index	12

Version
1.12

Ce guide décrit la gestion de la spécification Unicode par Python pour les données textuelles et explique les différents problèmes généralement rencontrés par les utilisateurs qui travaillent avec Unicode.

1 Introduction à Unicode

1.1 Définitions

Les programmes d'aujourd'hui doivent être capables de traiter une grande variété de caractères. Les applications sont souvent internationalisées pour afficher les messages et les résultats dans une variété de langues sélectionnables par l'utilisateur ; le même programme peut avoir besoin d'afficher un message d'erreur en anglais, français, japonais,

hébreu ou russe. Le contenu Web peut être écrit dans n'importe laquelle de ces langues et peut également inclure une variété de symboles émoji. Le type de chaîne de caractères de Python utilise le standard Unicode pour représenter les caractères, ce qui permet aux programmes Python de travailler avec tous ces différents caractères possibles.

Unicode (<https://www.unicode.org/>) est une spécification qui vise à lister tous les caractères utilisés par les langues humaines et à donner à chaque caractère son propre code unique. Les spécifications Unicode sont continuellement révisées et mises à jour pour ajouter de nouvelles langues et de nouveaux symboles.

Un **caractère** est le plus petit composant possible d'un texte. « A », « B », « C », etc. sont tous des caractères différents. Il en va de même pour « È » et « Í ». Les caractères varient selon la langue ou le contexte dont vous parlez. Par exemple, il y a un caractère pour « Chiffre Romain Un » (*Roman Numeral One*), « I », qui est séparé de la lettre majuscule « I ». Ils se ressemblent généralement, mais ce sont deux caractères différents qui ont des significations différentes.

Le standard Unicode décrit comment les caractères sont représentés par les **points de code**. Une valeur de point de code est un nombre entier compris entre 0 et 0x10FFFF (environ 1,1 million de valeurs possibles, le **nombre de valeurs réellement assignées** est inférieur à ce nombre). Dans le standard et dans le présent document, un point de code est écrit en utilisant la notation U+265E pour désigner le caractère avec la valeur 0x265e (9 822 en décimal).

La standard Unicode contient de nombreux tableaux contenant la liste des caractères et des points de code correspondants :

```
0061      'a'; LATIN SMALL LETTER A
0062      'b'; LATIN SMALL LETTER B
0063      'c'; LATIN SMALL LETTER C
...
007B      '{'; LEFT CURLY BRACKET
...
2167      'Ⅷ'; ROMAN NUMERAL EIGHT
2168      'Ⅸ'; ROMAN NUMERAL NINE
...
265E      '♞'; BLACK CHESS KNIGHT
265F      '♟'; BLACK CHESS PAWN
...
1F600     '😄'; GRINNING FACE
1F609     '😏'; WINKING FACE
...
```

À proprement parler, ces définitions laissent entendre qu'il est inutile de dire « c'est le caractère U+265E ». U+265E est un point de code, qui représente un caractère particulier ; dans ce cas, il représente le caractère « BLACK CHESS KNIGHT », « ♞ ». Dans des contextes informels, cette distinction entre les points de code et les caractères sera parfois oubliée.

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.

1.2 Encodages

Pour résumer la section précédente : une chaîne Unicode est une séquence de points de code, qui sont des nombres de 0 à 0x10FFFF (1 114 111 en décimal). Cette séquence de points de code doit être stockée en mémoire sous la forme d'un ensemble de **unités de code**, et les **unités de code** sont ensuite transposées en octets de 8 bits. Les règles de traduction d'une chaîne Unicode en une séquence d'octets sont appelées un **encodage de caractères** ou simplement un **encodage**.

Le premier encodage auquel vous pouvez penser est l'utilisation d'entiers 32 bits comme unité de code, puis l'utilisation de la représentation des entiers 32 bits par le CPU. Dans cette représentation, la chaîne « Python » ressemblerait à ceci :

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*).

Par conséquent, cet encodage n'est pas très utilisé et d'autres encodages, plus efficaces et pratiques comme UTF-8, sont plutôt choisis.

UTF-8 est l'un des encodages les plus couramment utilisés et Python l'utilise souvent par défaut. UTF signifie « Unicode Transformation Format » (format de transformation Unicode) et « 8 » signifie que des nombres à 8 bits sont utilisés dans l'encodage (il existe également des codages UTF-16 et UTF-32, mais ils sont moins souvent utilisés que UTF-8). UTF-8 utilise les règles suivantes :

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. Une chaîne Unicode est transformée en une séquence d'octets qui contient des octets zéro uniquement lorsqu'ils représentent le caractère nul (U+0000). Cela signifie que les chaînes UTF-8 peuvent être traitées par des fonctions C telles que `strcpy()` et envoyées par des protocoles pour qui les octets zéro signifient forcément la fin de chaîne.
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.
6. UTF-8 est un encodage orienté octet. L'encodage spécifie que chaque caractère est représenté par une séquence spécifique d'un ou plusieurs octets. Ceci permet d'éviter les problèmes d'ordre des octets qui peuvent survenir avec les encodages orientés entiers (*integer*) ou orientés mots processeurs (*words*), comme UTF-16 et UTF-32, où la séquence des octets varie en fonction du matériel sur lequel la chaîne a été encodée.

1.3 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.

Sur la chaîne Youtube *Computerphile*, Tom Scott parle brièvement de [l'histoire d'Unicode et d'UTF-8](#) (9 minutes et 36 secondes).

Pour aider à comprendre le standard, Jukka Korpela a écrit [un guide d'introduction](#) à la lecture des tables de caractères Unicode (ressource en anglais).

Un autre [bon article d'introduction](#) a été écrit par Joel Spolsky. Si cette présente introduction ne vous a pas clarifié les choses, vous devriez essayer de lire cet article-là avant de continuer.

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*

Depuis Python 3.0, le type `str` du langage contient des caractères Unicode, c'est-à-dire n'importe quelle chaîne créée à l'aide de "unicode déchire !", 'unicode déchire !' ou la syntaxe à triples guillemets est enregistrée comme 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")
```

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 CHARACTER), 'ignore' (« ignorer » : n'inclut pas le caractère dans le résultat Unicode) ou 'backslashreplace' (« remplacer avec anti-slash » : 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:
    invalid start byte
>>> b'\x80abc'.decode("utf-8", "replace")
'\ufffdabc'
>>> b'\x80abc'.decode("utf-8", "backslashreplace")
'\\x80abc'
>>> b'\x80abc'.decode("utf-8", "ignore")
'abc'
```

Les encodages sont spécifiés sous forme de chaînes de caractères contenant le nom de l'encodage. Python est livré avec une centaine d'encodages différents ; voir la référence de la bibliothèque Python sur les encodages standards pour une liste. Certains encodages ont plusieurs noms ; par exemple, 'latin-1', 'iso_8859_1' et '8859' sont tous synonymes du même encodage.

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 'replace' (qui dans ce cas insère un point d'interrogation au lieu du caractère non encodable), 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
```

(suite sur la page suivante)

```
... #          ^^^^^ 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

La spécification Unicode inclut une base de données d'informations sur les points de code. Pour chaque point de code défini, l'information comprend le nom du caractère, sa catégorie, la valeur numérique s'il y a lieu (pour les caractères représentant des concepts numériques tels que les chiffres romains, les fractions telles qu'un tiers et quatre cinquièmes, etc.). Il existe également des propriétés liées à l'affichage, telles que l'utilisation du point de code dans un texte bidirectionnel.

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=" ")
    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-espçant », 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 Comparaison de chaînes de caractères

Unicode ajoute une certaine complication à la comparaison des chaînes de caractères, car le même jeu de caractères peut être représenté par différentes séquences de points de code. Par exemple, une lettre comme « ê » peut être représentée comme un point de code unique U+00EA, ou comme U+0065 U+0302, qui est le point de code pour « e » suivi d'un point de code pour COMBINING CIRCUMFLEX ACCENT. Celles-ci produisent le même résultat lorsqu'elles sont affichées, mais l'une est une chaîne de caractères de longueur 1 et l'autre de longueur 2.

Un outil pour une comparaison insensible à la casse est la méthode `casefold()` qui convertit une chaîne en une forme insensible à la casse suivant un algorithme décrit par le standard Unicode. Cet algorithme a un traitement spécial pour les caractères tels que la lettre allemande « ß » (point de code U+00DF), qui devient la paire de lettres minuscules « ss ».

```
>>> street = 'Gürzenichstraße'
>>> street.casefold()
'gürzenichstrasse'
```

A second tool is the `unicodedata` module's `normalize()` function that converts strings to one of several normal forms, where letters followed by a combining character are replaced with single characters. `normalize()` can be used to perform string comparisons that won't falsely report inequality if two strings use combining characters differently :

```
import unicodedata

def compare_strs(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(s1) == NFD(s2)

single_char = 'ê'
multiple_chars = '\N{LATIN SMALL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}'
print('length of first string=', len(single_char))
print('length of second string=', len(multiple_chars))
print(compare_strs(single_char, multiple_chars))
```

Si vous l'exécutez, cela affiche :

```
$ python compare-strings.py
length of first string= 1
length of second string= 2
True
```

Le premier argument de la fonction `normalize()` est une chaîne de caractères donnant la forme de normalisation désirée, qui peut être une de celles-ci : 'NFC', 'NFKC', 'NFD' et 'NFKD'.

La norme Unicode spécifie également comment faire des comparaisons insensibles à la casse :

```
import unicodedata

def compare_caseless(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(NFD(s1).casefold()) == NFD(NFD(s2).casefold())

# Example usage
single_char = 'ê'
multiple_chars = '\N{LATIN CAPITAL LETTER E}\N{COMBINING CIRCUMFLEX ACCENT}'

print(compare_caseless(single_char, multiple_chars))
```

This will print `True`. (Why is `NFD()` invoked twice? Because there are a few characters that make `casefold()` return a non-normalized string, so the result needs to be normalized again. See section 3.13 of the Unicode Standard for a discussion and an example.)

2.6 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]`.

2.7 Références

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

- [Processing Text Files in Python 3](#), par Nick Coghlan.
- [Pragmatic Unicode](#), une présentation *PyCon* 2012 par 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+FEFF` 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*.

Dans certains cas, il est également d'usage d'utiliser une *BOM* au début des fichiers encodés en UTF-8 ; le nom est trompeur puisque l'UTF-8 ne dépend pas de l'ordre des octets. La marque annonce simplement que le fichier est encodé en UTF-8. Pour lire ces fichiers, utilisez le codec `utf-8-sig` pour sauter automatiquement la marque si elle est présente.

3.1 Noms de fichiers Unicode

La plupart des systèmes d'exploitation couramment utilisés aujourd'hui prennent en charge les noms de fichiers qui contiennent des caractères Unicode arbitraires. Habituellement, ceci est implémenté en convertissant la chaîne Unicode en un encodage qui varie en fonction du système. Aujourd'hui, Python converge vers l'utilisation d'UTF-8 : Python sous MacOS utilise UTF-8 depuis plusieurs versions et Python 3.6 sous Windows est passé à UTF-8 également. Sur les systèmes Unix, il n'y aura un encodage pour le système de fichiers que si vous avez défini les variables d'environnement `LANG` ou `LC_CTYPE` ; sinon, l'encodage par défaut est 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.

La fonction `os.listdir()` renvoie des noms de fichiers, ce qui soulève un problème : doit-elle renvoyer la version Unicode des noms de fichiers ou doit-elle renvoyer des chaînes d'octets contenant les versions encodées ? `os.listdir()` peut faire les deux, selon que vous fournissez le chemin du répertoire en chaîne d'octets ou en chaîne Unicode. Si vous passez une chaîne Unicode comme chemin d'accès, les noms de fichiers sont décodés en utilisant l'encodage du système de fichiers et une liste de chaînes Unicode est renvoyée, tandis que passer un chemin d'accès en chaîne d'octets renvoie les noms de fichiers comme chaîne d'octets. Par exemple, en supposant que l'encodage par défaut du système de fichiers est UTF-8, exécuter le programme suivant :

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

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

produit la sortie suivante :

```
$ python listdir-test.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.

Notez que, dans la plupart des cas, il convient de vous en tenir à l'utilisation d'Unicode avec ces *APIs*. Les *API* d'octets ne devraient être utilisées que sur les systèmes où des noms de fichiers non décodables peuvent être présents. Cela ne concerne pratiquement que des systèmes Unix maintenant.

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)
```

Le gestionnaire d'erreurs `surrogateescape` décode tous les octets non-ASCII comme points de code dans une plage spéciale allant de U+DC80 à U+DCFF. Ces points de code redeviennent alors les mêmes octets lorsque le gestionnaire d'erreurs `surrogateescape` est utilisé pour encoder les données et les réécrire.

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.

Merci aux personnes suivantes qui ont noté des erreurs ou qui ont fait des suggestions sur cet article : Éric Araujo, Nicholas Bastin, Nick Coghlan, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Terry J. Reedy, Serhiy Storchaka, Eryk Sun, Chad Whitacre, Graham Wideman.

Index

P

Python Enhancement Proposals

PEP 263, [6](#)