



L’encoding en Python, une bonne fois pour toute

This entry was posted in [Programmation](#) and tagged [ascii](#) [encoding](#) [python](#) [unicode](#) [utf8](#) on 21/04/2013 by Sam

J'avais oublié la zik, je rajoute:



Vous avez tous un jour eu l’erreur suivante :

```
UnicodeDecodeError: 'machine' codec can't decode character 'trucmuche'
```

Et là, pour vous en sortir, vous en avez chié des ronds de pâté.

Le problème vient du fait que la plupart du temps, ignorer l’encoding marche : nous travaillons dans des environnements homogènes et toujours avec des données dans le même format, ou un format plus ou moins compatible.

Mais le texte, c’est compliqué, terriblement compliqué, et le jour où ça se gâte, si vous ne savez pas ce que vous faites, vous ne vous en sortirez pas.

C’est d’autant plus vrai en Python car :

- Par défaut, Python plante sur les erreurs d’encoding là où d’autres langages (comme le PHP) se débrouillent pour vous sortir un truc (qui ne veut rien dire, qui peut corrompre toute votre base de données, mais qui ne plante pas).
- Python est utilisé dans des environnements hétérogènes. Quand vous codez en JS sur le navigateur, vous n’avez presque jamais à vous soucier de l’encoding : le browser gère quasiment tout pour vous. En Python dès que vous allez lire un fichier et l’afficher dans un terminal, cela fait potentiellement 3 encoding différents.
- Python 2.7 a des réglages par défaut très stricts, et pas forcément adaptés à notre informatique moderne (fichier de code en ASCII par exemple).

A la fin de cet article, vous saurez vous sortir de toutes les situations merdiques liées aux encodages.

Règle numéro 1 : Le texte brut n’existe pas.

Quand vous avez du texte quelque part (un terminal, un fichier, une base de données…), il est forcément représenté sous forme de 0 et de 1.

La corrélation entre cette suite de 0 et de 1 et la lettre est faite dans un énorme tableau qui contient toutes les lettres d’un côté, et toutes les combinaisons de 0 et de 1 de l’autre. Il n’y a pas de magie. C’est un énorme tableau stocké quelque part dans votre ordinateur. **Si vous n’avez pas ce tableau, vous ne pouvez pas lire du texte. Même le texte le plus simple.**

Malheureusement, au début de l’informatique, **presque chaque pays a créé son propre tableau, et ces tableaux sont incompatibles entre eux** : pour la même combinaison de 0 et de 1, ils donnent un caractère différent voire rien du tout.

Mais où je suis tombé(e) là ?

 Télécharger cette page en PDF

DEVENEZ CONTRIBUTEUR

IMPERTINENT



Souscrire à nos conneries

Entrez votre adresse mail et vous recevrez une notification à chaque nouvel article.

Join 1,492 other subscribers


Email Address

J'adhère, bon dieu !

 TagCloud

Obin angularjs asyncio autobahn **bash** bitcoin **blog** cache comprehension-lists crossbar css cul dict **django** don encoding git http import ipython iterable **javascript** jquery linux **meta** mysql nginx nsfw pep8 pip poo **python** python 3 redis ruby shell sublime text ubuntu unicode unit tests unpacking virtualenv wamp web yield

Envoyez des sioux

 On adooooore les bitcoins :

19zAHPPuce4BAhsdy9KaFwVLurEJXMhMAN

 Nos projets

› Multiboards – l’actu geek fr en une page

› Obin – le pastebin chiffré

› All that counts – compteur pour jeux

› Django quicky – quick views for Django

› VizHash.js – Hash visuels

› Code des articles du blog

Ces tableaux, c'est ce qu'on appelle les encodings, et il y en a beaucoup. Voici la liste de ceux que Python gère :

```
>>> import encodings
>>> print ''.join('- ' + e + '\n' for e in sorted(set(encodings.alias
- ascii
- base64_codec
- big5
- big5hkscs
- bz2_codec
- cp037
- cp1026
- cp1140
- cp1250
- cp1251
- cp1252
- cp1253
- cp1254
- cp1255
- cp1256
- cp1257
- cp1258
- cp424
- cp437
- cp500
- cp775
- cp850
- cp852
- cp855
- cp857
- cp858
- cp860
- cp861
- cp862
- cp863
- cp864
- cp865
- cp866
- cp869
- cp932
- cp949
- cp950
- euc_jis_2004
- euc_jisx0213
- euc_jp
- euc_kr
- gb18030
- gb2312
- gbk
- hex_codec
- hp_roman8
- hz
- iso2022_jp
- iso2022_jp_1
- iso2022_jp_2
- iso2022_jp_2004
- iso2022_jp_3
- iso2022_jp_ext
- iso2022_kr
- iso8859_10
- iso8859_11
- iso8859_13
- iso8859_14
- iso8859_15
- iso8859_16
- iso8859_2
- iso8859_3
- iso8859_4
- iso8859_5
- iso8859_6
- iso8859_7
- iso8859_8
- iso8859_9
- johab
- koi8_r
- latin_1
- mac_cyrillic
- mac_greek
- mac_iceland
- mac_latin2
- mac_roman
- mac_turkish
- mbc
- ptcp154
- quopri_codec
- rot_13
- shift_jis
- shift_jis_2004
- shift_jisx0213
- tactis
- tis_620
- utf_16
```

› [Log in](#)

› [Entries RSS](#)

› [Comments RSS](#)

› [WordPress.org](#)



```
- utf_16_be
- utf_16_le
- utf_32
- utf_32_be
- utf_32_le
- utf_7
- utf_8
- uu_codec
- zlib_codec
```

Et certains ont plusieurs noms (des alias), donc on pourrait en compter plus:

```
>>> len(encodings.aliases.aliases.keys())
307
```

Quand vous affichez du texte sur un terminal avec un simple `print`, votre ordinateur va implicitement chercher le tableau qu’il pense être le plus adapté, et fait la traduction. Même pour le texte le plus simple. Même pour un espace tout seul.

Mais surtout, ça veut dire que votre propre code EST dans un encoding. Et vous DEVEZ savoir lequel.

Règle numéro 2 : utf8 est le langage universel, utilisez-le

Il existe un encoding qui essaye des regrouper toutes les langues du monde, et il s’appelle unicode. Unicode est un tableau gigantesque qui contient des combinaisons de 1 et de 0 d’un côté, et les caractères de toutes la langues possibles de l’autre : chinois, arabe, français, espagnol, russe...

Bon, il ne contient pas encore absolument tout, mais il couvre suffisamment de terrain pour éliminer 99.999999999% des problèmes de communications de texte entre machines dans le monde.

Le défaut d’Unicode est qu’il est plus lent et prend plus de place que d’autres représentations du même texte. Aujourd’hui le téléphone le plus pourri a 10 fois la puissance nécessaire, et ce n’est plus un souci : il peut être utilisé presque partout (sauf peut-être dans l’embarqué drastique) sans même réfléchir à la question. Tous les langages les plus importants, tous les services les plus importants, tous les logiciels les plus importants gèrent unicode.

Il y a plusieurs implémentations concrètes d’unicode, la plus célèbre est “UTF 8”.

Moralité, par défaut, utilisez utf-8.

Une fois, à l’entretien d’embauche, un mec m’avait reproché d’utiliser UTF8 parce que “ça posait des problèmes d’encoding”. Comprenez bien qu’utf-8 ne pose aucun problème d’encoding. Ce sont tous les autres codecs du monde qui posent des problèmes d’encoding. UTF-8 est certainement le seul à justement, ne poser aucun problème.

UTF 8 est le seul encoding vers lequel, aujourd’hui, on puisse convertir vers et depuis (pratiquement) n’importe quel autre codec du monde. C’est un espéranto. C’est une pierre de rosette. C’est au texte ce que l’or est à l’économie.

Si UTF8 vous pose “un problème d’encoding”, c’est que vous ne savez pas dans quel encoding votre texte est actuellement ou comment le convertir. C’est tout.

Il n’y a presque aucune raison de ne pas utiliser UTF8 aujourd’hui (à part sur des vieux systèmes ou des systèmes où les ressources sont tellement limitées que vous n’utiliseriez pas Python de toute façon).

Utilisez utf8. Partout. Tout le temps.

Si vous communiquez avec un système qui ne comprend pas UTF8, convertissez.

Mais gardez votre partie en UTF8.

Règle numéro 3 : il faut maîtriser l’encoding de son code

Le fichier dans lequel vous écrivez votre code est dans un encoding et ce n’est pas lié à votre OS. C’est votre éditeur qui s’en occupe. Apprenez à régler votre éditeur pour qu’il utilise l’encoding que vous voulez.

Et l’encoding que vous voulez est UTF8.

Si vous ne savez pas dans quel encoding est votre code, vous ne pouvez pas manipuler du texte et garantir l’absence de bug.

Vous ne POUVEZ PAS.



Regardez dans la doc de l'éditeur, dans l'aide ou tapez sur Google, mais faites le.

Puis il faut déclarer cet encoding à la première ligne de chaque fichier de code avec l'expression suivante :

```
# coding: encoding
```

Par exemple :

```
# coding: utf8
```

C'est une spécificité de Python : si l'encoding du fichier est différent de l'encoding par défaut du langage, il faut le déclarer sinon le programme plantera à la première conversion. En Python 2.7, l'encoding par défaut est ASCII, donc il faut presque toujours le déclarer. En Python 3, l'encoding par défaut est UTF8 et on peut donc l'omettre si on l'utilise. Ce que vous allez faire après la lecture de cet article.

Ensuite, il existe deux types de chaînes de caractères en Python :

- La chaîne de caractères encodée: type 'str' en Python 2.7, 'byte' en Python 3.
- La chaîne de caractères décodée: type 'unicode' en Python 2.7, et 'str' en python 3 (sic).

Illustration :

```
$ python2.7
Python 2.7.3 (default, Aug 1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information
>>> type('chaîne') # bits => encodée
<type 'str'>
>>> type(u'chaîne') # unicode => décodée
<type 'unicode'>
```

```
$ python3
Python 3.2.3 (default, Oct 19 2012, 20:10:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information
>>> type("chaîne") # unicode => décodée
<class 'str'>
>>> type(b"chaîne") # bits => encodée
<class 'bytes'>
```

Votre but, c'est de n'avoir dans votre code que des chaînes de type 'unicode'.

En Python 3, c'est automatique. Toutes les chaînes sont de type 'unicode' (appelé 'str' dans cette version, je sais, je sais, c'est confusionnant à mort) par défaut.

En Python 2.7 en revanche, il faut préfixer la chaîne par un `u`.

Donc, dans votre code, TOUTES vos chaînes doivent être déclarées ainsi :

```
u"votre chaîne"
```

Oui, c'est chiant. Mais c'est indispensable. Encore une fois, il n'y a pas d'alternative (dites le avec la voix de Thatcher si ça vous excite).

Si vous voulez, vous pouvez activer le comportement de Python 3 dans Python 2.7 en mettant ceci au début de CHACUN de vos modules :

```
from __future__ import unicode_literals
```

Ceci n'affecte que le fichier en cours, jamais les autres modules.

On peut le mettre au démarrage d'iPython également.

Je résume :

- Réglez votre éditeur sur UTF8.
- Mettez `# coding: utf8` au début de vos modules.
- Préfixez toutes vos chaînes de `u` ou faites `from __future__ import unicode_literals` en début de chaque module.

Si vous ne faites pas cela, votre code marchera. La plupart de temps. Et un jour, dans une situation particulière, il ne marchera plus. Plus du tout.

Oh, et ce n'est pas grave si vous avez d'anciens modules dans d'autres encodings. Tant que vous utilisez des objets 'unicode' partout, ils marcheront sans problème ensemble.

Règle numéro 4 : décoder toutes les entrées de votre programme

La partie difficile de ce conseil, c'est de savoir ce qu'est une entrée.

Je vais vous donner une définition simple : **tout ce qui ne fait pas partie du code de votre programme et qui est traité dans votre programme est une entrée.**

Le texte des fichiers, le nom de ces fichiers, le retour des appels système, le retour d'une ligne de commande parsée, la saisie utilisateur sur un terminal, le retour d'une requête SQL, le téléchargement d'une donnée sur le Web, etc.

Ce sont toutes des entrées.

Comme tous les textes du monde, les entrées sont dans un encoding. Et vous DEVEZ savoir lequel.

Comprenez bien, si vous ne connaissez pas l'encoding de vos entrées, ça marchera la plupart du temps, et un jour, ça va planter.

Il n'y a pas d'alternative (bis).

Or, **il n'y a pas de moyen de détecter un encoding de façon fiable.**

Donc, soit le fournisseur de la donnée vous donne cette information (settings dans la base de données, doc de votre logiciel, configuration de votre OS, spec du client, coup de fils au fournisseur...), soit vous êtes baisés.

On ne peut pas lire un simple fichier si on ne connaît pas son encoding. Point.

Si cela a marché jusqu'ici pour vous, c'est que vous avez eu de la chance : la plupart de vos fichiers étaient dans l'encoding de votre éditeur et de votre système. Tant qu'on travaille sur sa machine, tout va bien.

Si vous lisez une page HTML, l'encoding est souvent déclaré dans la balise META ou dans un header.

Si vous écrivez dans un terminal, l'encoding du terminal est accessible avec `sys.stdin.encoding`.

Si vous manipulez des noms de fichier, on peut récupérer l'encoding du file system en cours avec `sys.getfilesystemencoding()`.

Mais parfois il n'y a pas d'autres moyens d'obtenir cette information que de demander à la personne qui a produit la donnée. Parfois même, l'encoding déclaré est faux.

Dans tous les cas, vous avez besoin de cette information.

Et une fois que vous l'avez, il faut décoder le texte reçu.

La manière la plus simple de faire cela est :

```
your_string = your_string.decode('nom_du_codec')
```

Le texte sera de type 'str', et `decode()` retourne (si vous lui fournissez le bon codec ;-)), une version 'unicode'.

Exemple, obtenir une chaîne 'unicode' depuis une chaîne 'str' encodée en utf8 :

```
>>> une_chaine = 'Chaîne' # mon fichier est encodé en UTF8, donc la c
>>> type(une_chaine)
<type 'str'>
>>> une_chaine = une_chaine.decode('utf8')
>>> type(une_chaine)
<type 'unicode'>
```

Donc dès que vous lisez un fichier, récupérez une réponse d'une base de données ou parsez des arguments d'un terminal, appelez `decode()` sur la chaîne reçue.

Règle numéro 5 : encodez toutes les sorties de votre programme

La partie difficile de ce conseil, c'est de savoir ce qu'est une sortie.

Encore une fois, une définition simple : **toute donnée que vous traitez et qui va être lue par autre chose que votre code est une sortie.**

Un `print` dans un terminal est une sortie, un `write()` dans un fichier est une sortie, un `UPDATE` en SQL est une sortie, un envoi dans une socket est une sortie, etc.

Le reste du monde ne peut pas lire les objets 'unicode' de Python. Si vous écrivez ces objets dans un fichier, un terminal ou dans une base de données, Python va les convertir automatiquement en objet 'str', et l'encoding utilisé dépendra du contexte.

Malheureusement, il y a une limite à la capacité de Python à décider du bon encoding.

Donc, tout comme il vous faut connaître l’encoding d’un texte en entrée, il vous faut connaître l’encoding attendu par le système avec lequel vous communiquez en sortie : sachez quel est l’encoding du terminal, de votre base de données ou système de fichiers sur lequel vous écrivez.

Si vous ne pouvez pas savoir (page Web, API, etc), utilisez UTF8.

Pour ce faire, il suffit d’appelerz `encode()` sur tout objet de type ‘unicode’ :

```
une_chaine = une_chaine.encode('nom_du_codec')
```

Par exemple, pour convertir un objet ‘unicode’ en ‘str’ utf8:

```
>>> une_chaine = u'Chaîne'
>>> type(une_chaine)
<type 'unicode'>
>>> une_chaine = une_chaine.encode('utf8')
>>> type(une_chaine)
<type 'str'>
```

Résumé des règles

- 1. Le texte brut n’existe pas.
- 2. Utilisez UTF8. Maintenant. Partout.
- 3. Dans votre code, spécifiez l’encoding du fichier et déclarez vos chaînes comme ‘unicode’.
- 4. À l’entrée, connaissez l’encoding de vos données, et décodez avec `decode()` .
- 5. A la sortie, encodez dans l’encoding attendu par le système qui va recevoir la données, ou si vous ne pouvez pas savoir, en UTF8, avec `encode()` .

Je sais que ça vous démange de voir un cas concret, alors voici un pseudo programme (téléchargeable ici) :



```
# toutes les chaînes sont en unicode (même les docstrings)
from __future__ import unicode_literals

"""
    Un script tout pourri qui télécharge plein de page et les sauvegarde
    dans une base de données sqlites.

    On écrit dans un fichier de log les opérations effectuées.
"""

import re
import urllib2
import sqlite3

pages = (
    ('Snippets de Sebsauvage', 'http://www.sebsauvage.net/python/snippets/'),
    ('Top 50 de bashfr', 'http://danstonchat.com/top50.html'),
)

# création de la base de données
conn = sqlite3.connect(r"backup.db")
c = conn.cursor()

try:
    c.execute('''
        CREATE TABLE pages (
            id INTEGER PRIMARY KEY,
            nom TEXT,
            html TEXT
        )''')
except sqlite3.OperationalError:
    pass

log = open('backup.log', 'wa')

for nom, page in pages:

    # ceci est une manière très fragile de télécharger et
    # parser du HTML. Utilisez plutôt scrapy et BeautifulSoup
    # si vous faites un vrai crawler
    response = urllib2.urlopen(page)
    html = response.read(100000)

    # je récupère l'encoding à l'arrache
    encoding = re.findall(r'<meta.*?charset=["\']*?(.+?)["\']*>', html, re.I)

    # html devient de l'unicode
    html = html.decode(encoding)

    # ici je peux faire des traitements divers et variés avec ma chaîne
    # et en fin de programme...

    # la lib sqlite convertit par défaut tout objet unicode en UTF8
    # car c'est l'encoding de sqlite par défaut donc passer des chaînes
    # unicode marche, et toutes les chaînes de mon programme sont en unicode
    # grâce à mon premier import
    c.execute("""INSERT INTO pages (nom, html) VALUES (?, ?)""", (nom, html))

    # j'écris dans mon fichier en UTF8 car c'est ce que je veux pouvoir
    # plus tard
    msg = "Page '{}' sauvée\n".format(nom)
    log.write(msg.encode('utf8'))

    # notez que si je ne fais pas encode(), soit:
    # - j'ai un objet 'unicode' et ça plante
    # - j'ai un objet 'str' et ça va marcher mais mon fichier contiendra
    #   l'encoding de la chaîne initiale (qui ici serait aussi UTF8, mais ce n'est pas toujours le cas)

conn.commit()
c.close()

log.close()
```

Quelques astuces

Certaines bibliothèques acceptent indifféremment des objets 'unicode' et 'str' :

```
>>> from logging import basicConfig, getLogger
>>> basicConfig()
>>> log = getLogger()
>>> log.warn("Détecté")
WARNING:root:Détecté
>>> log.warn(u"Détecté")
WARNING:root:Détecté
```





D'autres ont besoin qu'on leur précise:

```
>>> import re
>>> import re
>>> re.search('é', 'télé')
<_sre.SRE_Match object at 0x7fa4d3f77238>
>>> re.search(u'é', u'télé', re.UNICODE)
<_sre.SRE_Match object at 0x7fa4d3f772a0>
```

Le module `re` par exemple aura des résultats biaisés sur une chaîne 'unicode' si on ne précise pas le flag `re.UNICODE`.

D'autres n'acceptent pas d'objet 'str':

```
>>> import io
>>> >>> io.StringIO(u'é')
<_io.StringIO object at 0x14a96d0>
>>> io.StringIO(u'é'.encode('utf8'))
Traceback (most recent call last):
  File "<ipython-input-5-16988a0d4ac4>", line 1, in <module>
    io.StringIO('é'.encode('utf8'))
TypeError: initial_value must be unicode or None, not str
```

D'autres encore n'acceptent pas d'objet 'unicode':

```
>>> import base64
>>> base64.encodestring('é'.encode('utf8'))
'w6k=\n'
>>> base64.encodestring(u'é')
Traceback (most recent call last):
  File "<ipython-input-3-1714982ca68e>", line 1, in <module>
    base64.encodestring('é')
  File "/usr/lib/python2.7/base64.py", line 315, in encodestring
    pieces.append(binascii.b2a_base64(chunk))
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in p
```

Cela peut être pour des raison de performances (certaines opérations sont plus rapides sur un objet 'str'), ou pour des raisons historiques, d'ignorance ou de paresse.

Vous ne pouvez pas le deviner à l'avance. Souvent c'est marqué dans la doc, sinon il faut tester dans le shell.

Une bibliothèque bien faite demandera de l'unicode et vous retournera de l'unicode, vous libérant l'esprit. Par exemple, requests et l'ORM Django le font, et communiquent avec le reste du monde (en l'occurrence le Web et les bases de données) dans le meilleur encoding possible automatiquement de manière transparente. Quand c'est possible bien entendu, parfois il faudra forcer l'encoding car le fournisseur de votre donnée déclare le mauvais. Vous n'y pouvez rien, c'est pareil pour tous les langages du monde.

Enfin il existe des raccourcis pour certaines opérations, utilisez-les autant que possible. Par exemple, pour lire un fichier, au lieu de faire un simple `open()`, vous pouvez faire :

```
from codecs import open

# open() de codec à exactement La même API, y compris avec "with"
f = open('fichier', encoding='encoding')
```

Les chaînes récupérées seront automatiquement sous forme d'objet 'unicode' au lieu d'objet 'str' qu'il vous aurait fallu convertir à la main.

Les outils de la dernière chance

Je vous ai menti, si vous ne connaissez pas l'encoding de vos entrées ou de vos sorties, il vous reste encore quelques options.

Sachez cependant que ces options sont des hacks, des trucs à tenter quand tout ce qui a été décrit plus haut a foiré.

Si vous faites bien votre boulot, ça ne doit pas arriver souvent. Une à deux fois max dans votre année, sauf environnement de travail très très merdique.

D'abord, parlons de l'entrée.

Si vous recevez un objet et qu'il vous est impossible de trouver l'encoding, vous pouvez forcer un décodage imparfait avec `decode()` en spécifiant le paramètre `error`.

Il peut prendre les valeurs suivantes :



- 'strict' : lever une exception en cas d'erreur. C'est le comportement par défaut.
- 'ignore' : tout caractère qui provoque une erreur est ignoré.
- 'replace' : tout caractère qui provoque une erreur est remplacé par un point d'interrogation.

```
>>> print 'Père Noël'.decode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position
>>> print 'Père Noël'.decode('ascii', errors='ignore')
Pre Nol
>>> print 'Père Noël'.decode('ascii', errors='replace')
P?re No?l
```

Mozilla vient également à la rescousse avec sa lib chardet qu'il faut donc installer :

```
pip install chardet
```

Et qui TENTE (du verbe ‘tenter’, “qui essaye”, et qui donc peut échouer et se tromper) de détecter l’encoding utilisé.

```
>>> chardet.detect(u'Le Père Noël est une ordure'.encode('utf8'))
{'confidence': 0.8063275188616134, 'encoding': 'ISO-8859-2'}
>>> chardet.detect(u"Le Père Noël est une ordure j'ai dis enculé".enc
{'confidence': 0.87625, 'encoding': 'utf-8'}
```

Cela marche pas trop mal, mais n’attendez pas de miracles. Plus il y a de texte, plus c’est précis, et plus le paramètre confidence est proche de 1.

Parlons maintenant de la sortie, c’est à dire le cas où le système qui va recevoir vos données est une grosse quiche qui plante dès qu’on lui donne autre chose que de l’ASCII.

Je ne veux balancer personne, mais mon regard se tourne vers l’administration américaine. Subtilement. De manière insistante.

D’abord, encode() accepte les mêmes valeurs pour errors que decode(). Mais en prime, il accepte 'xmlcharrefreplace', très pratique pour les fichiers XML :

```
>>> u"Et là-bas, tu vois, c'est la coulée du grand bronze".encode('as
"Et l&#224;-bas, tu vois, c'est la coul&#233;e du grand bronze"
```

Enfin, on peut essayer d’obtenir un texte potable en remplaçant les caractères spéciaux par leur équivalent ASCII le plus proche.

Avec l’alphabet latin, c’est très facile :

```
>>> unicodedata.normalize('NFKD', u"éçûö").encode('ascii', 'ignore')
'ecuo'
```

Pour des trucs plus avancés comme le cyrillique ou le mandarin, il faut installer unicode :

```
pip install unicode
```

```
>>> from unicode import unicode
>>> print unicode(u"En russe, Moscou s'écrit Москва")
En russe, Moscou s'ecrit Moskva
```

Partager:


[Tweet](#)

[Share 28](#)

[Email](#)

[More](#)

119 thoughts on “L’encoding en Python, une bonne fois pour toute”



roro

21/04/2013 at 08:31

Merci, merci, merci, et...MERCI.

Ta statue sera équestre. Pas moins.





Sam

Post author

21/04/2013 at 09:16

Fait gaffe aux nombres de pieds en l'air pour le cheval. J'ai pas envie qu'on me prenne pour un macabé.

Pour la matière, le bronze est mon meilleur profile.



k3c

21/04/2013 at 13:02

s/et le jour où ça se gatte/et le jour où ça se gâte
s/à ou d'autres langage/à où d'autres langages
s/Et certains on plusieurs noms/Et certains ont plusieurs noms
s/c'est ce qu'on appelle les encodings/c'est ce qu'on appelle les encodings
s/Certaines bibliothèque /Certaines bibliothèques

Le sodomiseur de coléoptères :-)

Merci pour l'article, je pensais pas trop mal connaître le sujet, mais visiblement non.



Sam

Post author

21/04/2013 at 13:18

Bah tu sais moi je me dis après chaque article que j'ai bien tout relu et qu'il reste pas de faute. Mais visiblement non.



fab

21/04/2013 at 17:50

Franchement merci.

Pour avoir lu plein de choses sur le sujet, mal compris, confondu le vocabulaire, oublié le lendemain, etc... c'est l'article qui explique le mieux et le plus concrètement l'encoding. C'est de la même veine que les articles sur la programmation orientée objet qui sont d'un niveau pédagogique inégalé parmi tout ce que j'ai pu lire. La théorie et les bases sont parfaitement expliquées avec toujours une application concrète. C'est limpide et efficace.

Cela fait un petit moment que je suis régulièrement votre blog, c'est une petite pépite, donc bravo et surtout un grand merci pour le partage car mine du rien c'est du boulot.



Sam

Post author

21/04/2013 at 18:07

\ o /



Lyyn

21/04/2013 at 20:33

Yeah, ça part dans mes favoris ! Merci pour l'article !



JeromeJ

21/04/2013 at 20:33

Premier commentaire constructif de la journée :

Vous avez déjà utilisé cette image d'entête d'article pour un autre article, c'est trop nul, vous perdez votre originalité.

<3



Sam

Post author

21/04/2013 at 20:39

Meeeeeeeeeeeeerde, j'ai eu un doute en plus.

EDIT: m'en fout, je l'ai changé.



JeromeJ

22/04/2013 at 00:58

Sinon plus sérieusement ... c'est un peu perturbant, vous parlez un peu des différences avec Python3 mais la majorité des exemples sont en Python2 et vu que les différences entre python2 et 3 (à ce niveau là en tout cas) sont super chiantes et prêtent à confusion, bah du coup c'est un peu plus dur de bien vous suivre partout (moi qui suis 100% Python3), dommage ... Article bien complet sinon :) Merci.

Ah et si l'un de vous à une solution simple pour faire en sorte que les encodings de stdout et stdin se mettent en UTF-8 par défaut, ça serait pas mal. (Surtout que ce sont des propriétés en read only donc, à part des méthodes bizarroïde au lancement de Python (en console), je ne sais même pas si y a moyen de les modifier en live).



JeromeJ

22/04/2013 at 01:11

Au fait, ~~je veux~~ j'aurais bien voulu un tampon :((tout ça pour ça bitch !)



Max

22/04/2013 at 07:26

le tampon ça se mérite jeune homme ! N'est pas tamponné qui veut.



Sam

Post author

22/04/2013 at 07:30

@JeromeJ: pour Python3, c'est pareil, exactement le même article sauf que:

- le type 'unicode' est appelé 'str' et est celui créé par défaut quand on fabrique une chaîne
- le type décodé est appelé 'byte'
- pas besoin de déclarer "# -*- coding: utf-8 -*-" si ton fichier est en UTF8 car c'est maintenant la valeur par default

C'est tout. Tout le reste est pareil.

Quand à s'assurer que toute écriture sur sdtout est en UTF8, c'est sans doute possible en faisant un truc du genre :

```
import sys
from codecs import getwriter

sys.__stdout__ = getwriter('utf8')(sys.stdout.buffer)
```

Mais je ne le recommanderais pas car :

- ça remplace stout pour tout le code, y compris les libs tierces parties et qui sait ce que ça peut provoquer
- ça rend le code importable (stdout n'est pas en utf8 sous windows, et ça printera un truc illisible)
- ton script peut changer de context en cours d'exécution, et sdtout ne sera alors plus le même, avec potentiellement un encoding différent

D'une manière générale il vaut mieux spécifier l'encoding par défaut directement sur les fonctions qui font l'opération d'écriture. Par exemple:



```
import sys
from codecs import getwriter
from functools import partial

p = partial(print, file=codecs.getwriter('utf8')(sys.s
p('yéééééééé') # cette fonction print toujours sur std
```

Une alternative est de setter la bonne variable d’environnement pour que Python remplisse stdout avec l’encoding que l’on souhaite. Par exemple sous Ubuntu :

<http://drj11.wordpress.com/2007/05/14/python-how-is-sysstdoutencoding-chosen/>

(ou PYTHONIOENCODING=utf-8:surrogateescape python somescript.py selon les versions)

Cela conditionne par ailleurs le résultat de :

```
>>> import locale
>>> locale.getdefaultlocale()[1]
```



JeromeJ

22/04/2013 at 07:48

@Max: J'aurais essayé !

@Sam: Bien ce que je craignais ... Pas mal l'idée de wrapper print avec functools.partial, j'aime !

(C'est bien le prob c'est que je travaille à moitié sous Windows)



Sam

Post author

22/04/2013 at 08:07

Le terminal de windows n’est pas configuré pour afficher de l’UTF8, Python ne pourra rien y changer. Il faut soit y printer l’encoding qu’il utilise actuellement (et se limiter aux caractères qu’il accepte du coup). On peut le trouver en faisant (il me semble):

```
>>> import locale
>>> locale.getdefaultlocale()
('fr_FR', 'UTF-8')
```

Mais print devrait se charger de le faire automatiquement en Python3 donc ça devrait pas poser de problème. Simplement tu oublis les caractères chinois.

Soit setter l’encoding avant avec une commande :

<http://stackoverflow.com/questions/388490/unicode-characters-in-windows-command-line-how>

Ce n’est pas particulièrement lié à Python, c’est pareil dans tous les langages sous Windows car cet OS n’a pas adopté utf8 par default.



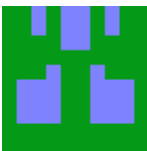
desfrenes

22/04/2013 at 08:42

“Par défaut, Python plante sur les erreurs d’encoding là ou d’autres langage (comme le PHP) se débrouillent pour vous sortir un truc (qui ne veut rien dire, qui peut corrompre toute votre base de données, mais qui ne plante pas).”

Ouais... bah vaut mieux que ça plante, au moins ça donne l’occasion de réparer proprement.

Merci pour cet excellent résumé.



AxelF

22/04/2013 at 08:54

Il y a encore des choses plus rigolotes à faire avec l’encoding. Par exemple ce fichier est tout à fait valide :

```
# -*- coding: rot13 -*-  
cevag h'Obawbhe fnz'
```



j'avais découvert ça il y a quelques temps sur linuxfr, avec même en prime l'encodage en brainfuck, mais on peut également imaginer encoder son fichier en whitespace ou tout autre encodage hyper pratique ...



Sam Post author

22/04/2013 at 09:09

Tristement, ça ne marche plus avec Python 3. On pouvait aussi utiliser base64, zlib, etc.



contre

22/04/2013 at 09:31

Uhu : <http://sebsauvage.net/links/?Q3zlyg>
Coïncidence ?

C'est la première fois que j'ai l'espoir de comprendre et d'utiliser correctement les fichues chaînes de caractère en python. Chapeau les mecs ! Mais je flippe quand même de modifier mon code sur ce sujet, qui fonctionne actuellement à peu près.

Par rapport à codec.open, il y a ça dans la doc :

Note
Files are always opened in binary mode, even if no binary mode was specified. This is done to avoid data loss due to encodings using 8-bit values. This means that no automatic conversion of ‘\n’ is done on reading and writing.

Ça peut être emmerdant ça ? Il y a un moyen simple de faire la conversion à la main ?



Sam Post author

22/04/2013 at 11:44

Je dirais qu'un bon strip() fera bien l'affaire :-)



contre

22/04/2013 at 12:19

En effet, ça marche bien à la lecture, mais du coup pas à l'écriture. Y'a plus qu'à utiliser la version standard sans oublier de convertir, j'ai l'impression.



contre

22/04/2013 at 13:38

Autre différence python2/3 : les exceptions. En python3 on peut leur passer un message en unicode, alors qu'en python2 il faut le .encode("utf-8). Et ça chie avec les docstrings, après...



YvesD

22/04/2013 at 14:48

Bel article, bravo, j'aime bien la structuration en entrée (eh eh les entrées, règle #4, plat (le traitement python), dessert (les sorties, règle #5).

Cependant je vois souvent comme directivepython, les 2 écritures pour UTF8:



*- coding: utf8 *-
ou
*- coding: utf-8 *-



Est-ce la même chose ? Laquelle est à privilégier ?



Sam Post author
22/04/2013 at 14:57
C'est kif kif.



Orlanth
22/04/2013 at 15:14
Copié collé direct dans mes notes, à côté de :
http://sebsauvage.net/python/charsets_et_encoding.html



Gring
23/04/2013 at 09:49
Merci pour cet article très instructif !



mentat
23/04/2013 at 19:31
stdin et pas stding dans :
sys.(stding|stdout).encoding.

A part ça, rien à redire, c'est nickel comme d'hab !



Sam Post author
23/04/2013 at 21:38
C'est l'acceng, c'est pour ça.



Kyoku57
05/05/2013 at 10:42
Alors là, très bon article !

Quelque soit le langage, le problème le plus récurrent est toujours celui de la manipulation des différents types d'encoding.

Je ne te parle pas de ceux qui développent sous Windows avec un CP-1252 par défaut dans tous les logiciels et qui même avec un réglage de l'IDE finit un jour où l'autre par te ressortir l'encoding par défaut genre .. à la réouverture.

Bref, éternel problème que tu as très bien mis en évidence. Tout le problème est d'origine interface chaise-clavier. Il faut se sortir les doigts du boule pour faire du propre.



yuiio
05/08/2013 at 14:26
Juste merci



A_Vgyle
27/09/2013 at 20:25
Merci grandement!
2 jours que je plante sur ces %?!/%££!! d'encoding, et en 15 minutes... Réglé
Thanx
Continue!!!





Max

01/10/2013 at 10:27

C'est laborieux chez leaseweb :/

Ensuite ça a été un problème de varnish qui s'est bien lancé quand leaseweb a reboot les VPS mais qui pour une raison mystérieuse ne voulait rien savoir :/



Guts

16/10/2013 at 10:29

Merci pour l'article qui m'a bien aidé sur certains points de blocage mais il y a toujours quelque chose que je ne parviens pas à résoudre : le parsing (avec `os.walk`) de fichiers qui contiennent des accents. Ce qui me déroute surtout c'est la différence entre Windows et Ubuntu.

Dans mon programme j'ai une fonction (`li_geofiles()`) chargée de lister les fichiers compatibles (selon leur extension et l'existence de leurs dépendances) contenus dans une arborescence à partir d'un dossier indiqué par l'utilisateur (`foldertarget` retourné via `tkMessageBox.askdirectory`) et qui bloque sur les noms de fichiers qui contiennent un accent...mais seulement sur Ubuntu ! Ça fait 2 jours que je me flagelle là-dessus et j'ai essayé qq's solutions :

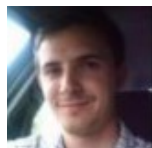
- un `try/except` sur le nom de fichier et réencodage à la volée (solution choisie) mais qui a le (gros) défaut de ne pas lister les fichiers avec accents (`os.path.isfile` ne les reconnaît pas...)
- passer `foldertarget` en unicode mais ça marche pô (idem avec `path.normpath(foldertarget)`)

Voici la fonction en question :

```
for root, dirs, files in walk(foldertarget):
    self.num_folders = self.num_folders + len(dirs)
    for f in files:
        try:
            unicode(path.join(root, f))
            full_path = path.join(root, f)
        except UnicodeDecodeError, e:
            full_path = path.join(root, f.decode('
            self.logger.error("%s:%s" % (e, f.deco
# Looping on files contained
if path.splitext(full_path.lower())[1] ==
    path.isfile('%s.dbf'.lower() % full_pa
    path.isfile('%s.shx'.lower() % full_pa
    path.isfile('%s.prj'.lower() % full_pa
        # add complete path of shapefile
        self.li_shp.append(full_path)
    elif path.splitext(full_path.lower())[1] =
        path.isfile(full_path[:-4]+ '.dat'.low
        path.isfile(full_path[:-4]+ '.map'.low
        path.isfile(full_path[:-4]+ '.id'.lowe
            # add complete path of MapInfo fil
            self.li_tab.append(full_path)
```

Si quelqu'un voit le problème...et surtout la solution !

PS : je suis géographe à la base donc pas vraiment un développeur... #indulgence



Adrien

04/11/2013 at 14:41

Un grand merci, enfin un tuto qui va à l'essentiel. J'ai enfin compris comment ça fonctionne.



Bromway

26/12/2013 at 22:43

Juste au Top!

Un super cadeau de Noel qui vient de m'offrir une ou deux nuit planche de moins



Réchère

20/01/2014 at 23:01

Article auquel je me réfère régulièrement, car à chaque fois que je résout mes problèmes d'encodage, je ne me souviens plus comment je les ai résolus et je dois refaire le raisonnement dans mon cerveau.

Juste pour info : y'a des fois, `sys.stdout.encoding` renvoie `None`.

Je suis en python 2.7 sous Windows (oui, je sais), quand j'exécute mon script dans la console, j'ai "cp850". Quand je l'exécute avec Ctrl-B dans Sublime Text, j'ai `None`.

Du coup, je fais le bourrin comme ça pour écrire sur `stdout` :

```
# récupération de l'encodage de la sortie standard.
encoding_out = sys.stdout.encoding
if encoding_out is None:
    encoding_out = "utf-8"

def pri(whatever):
    unicode_string = unicode(whatever)
    print unicode_string.encode(encoding_out, "replace")

pri("é mon cul cé du poulé.")
```



Réchère

21/01/2014 at 08:47

Cela va sans dire, mais je précise que dans mon code précédent, j'ai bien évidemment mis le "from __future__ import unicode_literals" au début de mes fichiers.

Du coup la chaîne "é mon cul cé du poulé" est de l'unicode.

Et ça marche, je vois mes accents, aussi bien dans la console DOS que dans la console Sublime Text.



Frederic

06/02/2014 at 21:55

Article salvateur!

Juste une question:

`os.mkdir(stringenutf8.encode(sys.getfilesystemencoding()))` ne marche pas sur windows avec python 2.7

Tu sais pourquoi?



Sam

Post author

07/02/2014 at 08:55

Parce qu'il y a plusieurs notions mélangées dans cette ligne.

`mkdir` a besoin d'une chaîne encodée dans l'encoding du file system.

`stringenutf8` est visiblement déjà encodée. On ne peut pas appelée "encode()" dessus.

Soit l'encoding du fs est utf8, auquel cas tu peux l'utiliser directement, soit il faut décoder la string vers unicode, et la réencoder vers l'encoding du fs.



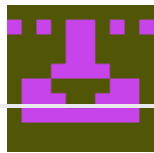
benjamin

04/04/2014 at 08:22

tu m'as sauvé la vie, 1 semaine à me battre sur un code car un `pu###` de "" dans l'adresse !!

bref, je te vénère. Merci pour cet excellent article en VF.

ps : j'aime bien l'exemple `str("Détecté")` ...



Goldy

17/04/2014 at 23:32

Une petite chose qui me semble avoir été omise dans l'explication, c'est que utf-8 est compatible avec ascii.

C'est à dire que les 128 premiers caractères de la table utf-8 correspondent à la table ascii (c'est d'ailleurs une des raisons pour laquelle utf-8 est utilisé aux détriments des autres unicodes qui sont utf-16 et utf-32).



fredtantini

18/04/2014 at 07:28

Je retombe dessus ; toujours très bon, comme d'habitude.

J'en profite pour corriger quelques typos :

Le texte sera de type 'str', et decode() retourne (si vous lui fournissez le bon codec ;-)), une version 'unicode'.

-> soit virer la virgule, soit changer les parenthèses en virgules.

un envoie dans une socket

-> envoi

A la sortie

-> À

Les outils de la dernières chance

-> dernière

cyrilique

-> cyrillique

Merci à vous et continuez en restant comme vous êtes.



e-jambon

20/04/2014 at 01:15

Il eut été plaisant que je commençasse à lire ce blog il y a quelques mois, avant de décider d'apprendre ruby&rails.

C'est fou ce que python ressemble à ruby, soit dit en passant.

Peut être un chouilla moins élégant, mais moins tortueux à relire quand on connaît pas...la preuve j'arrive à comprendre presque tout sans faire trop d'efforts.

Given l'am as bright as a dark night...excellents articles très bien rédigés. Un grand merci.

@Goldy merci pour la précision, ça me chiffonnait justement.



Sam

Post author

20/04/2014 at 12:11

@Goldy: j'étais persuadé de l'avoir mis, je deviens gaga.



Jean-Marie Sézérat

02/05/2014 at 12:52

Vrabo

Disons tous en chœur : “àèèùçÀÉÈÙÇ” – non : en chœur, c'est quand même plus joli...



kontre

05/05/2014 at 08:10

Dans python3, on peut passer un encoding à la fonction open(), le décodage se fera de manière transparente. Super pratique !

C'est possible en python2 avec le module coding, mais celui-ci, ne gère pas les différentes fin de ligne, ce qui est bof pour un truc sensé être plus compatible.





vecenzomene

07/05/2014 at 14:37

bonjour,
si par exemple je telecharge la page de google qui est securisé et
ya du sharabiaya dans le code source et qui est en binaire
comment faire pour convertir en utf-8 sachant que j'ai une erreur de
conversion quand j'essai et puis comment faire pour lire les vrai
contenue par exemple les resulats des recherche avec les liens et
non le charabiya de truc securisé



Sam

Post author

09/05/2014 at 09:38

Je ne connais aucune page de google sécursé avec un code
source en binaire.

Quand je vais sur google.com, je vois bien le code source :

http://0bin.net/paste/9-
wDHADDZvekQFyC#G1VTtoTRNQslSE7ZaWWYI1eAK5l8k1sEjNy
aFWej+DfQ=

C'est beaucoup de javascript compressé, mais c'est un code
source en clair.

Le charset n'est pas déclaré dans la page, mais par les headers :

```
> wget --server-response --spider https://google.com
Mode « spider » activé. Vérification de l'existence d'
--2014-05-09 16:06:05-- https://google.com/
Résolution de google.com (google.com)... 74.125.135.100,
Connexion à google.com (google.com)|74.125.135.100|:44
requête HTTP transmise, en attente de la réponse...
HTTP/1.1 302 Found
Location: https://www.google.co.th/?gws_rd=cr&ei=f5p
Cache-Control: private
Content-Type: text/html; charset=UTF-8
```

Donc c'est bien de l'utf8.

Obtenir les résultats de la recherche plutôt que le code n'a rien à
voir avec cela. Ce n'est pas une question d'encoding, ni de binaire,
c'est simplement le niveau de fonctionnalité que google choisit
d'utiliser en fonction du client. Si tu te fais passer pour lynx par
exemple, google n'utilisera pas javascript et tu récupéreras du
HTML pure, plus facile à parser. Par exemple, avec la lib requets :

```
requests.get("https://google.com/search?q=sametmax", h
```

Et ça me file du HTML que je peux lire avec beautifulsoup.



thieery

20/05/2014 at 22:51

Bonjour,
je cherche à lire le fichier dump windows et rien à faire, avez-vous
une idée merci



Sam

Post author

21/05/2014 at 05:34

Il faut manger des crêpes au chocolat. Tout va mieux avec des
crêpes au chocolat.



Heavy27z

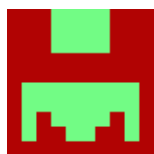
13/06/2014 at 09:14

Parfait cet article pour se dépatouiller quand on y comprend
queudal :).
Merci pour ce gros travail. En 20 minutes de lecture, ce blog me fait
gagner à chaque fois minimum 2h. Faudrait que j'vous verse une
partie de mon salaire...



Petit détail concernant l'article : Perso, j'ai du mal avec la notion d'encode et de decode (dans quel sens est-ce que ça marche), puisque finalement le texte de brute n'existe pas comme c'est dit, à quoi correspondent les données que l'on a de part et d'autre d'une chaîne str ou byte (pas simple de se faire comprendre, bref).

J'ai trouvé des éléments de réponses dans un article détaillé de la doc python : <https://docs.python.org/3/howto/unicode.html> (Existe aussi pour python 2)



Sam

Post author

13/06/2014 at 10:18

Le texte est un flux de bits. Ce flux de bits est dans un encoding. Quand on décode, on récupère un objet python qui peut être traduit vers n'importe quel autre flux de bits. Ce qu'il y a dans cet objet Python n'est pas important. Il faut imaginer qu'il est neutre. Decoder fait passer du flux de bits à cet objet Python, encoder fait passer de cet objet python aux flux de bits.



Heavy27z

13/06/2014 at 10:46

Merci d'avoir pris le temps de répondre, la doc python m'avait fait comprendre quelque chose de ce genre, mais là c'est très clair :).



gmarceau

01/07/2014 at 20:43

Merci pour cet article clair et utile.



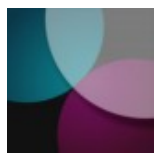
Mars Prosac

19/07/2014 at 22:19

Petit bémol à l'article : *“En Python 2.7, l'encoding par défaut est ASCII, donc il faut presque toujours le déclarer”*

Oui, il faut le déclarer si on souhaite coder autrement qu'en ascii, et dans ce cas, ça revient à dire “si on souhaite coder des strings autre qu'en ascii”. Autre précision qu'il ne me semble pas avoir lu : UTF8 est compatible ASCII (pas le contraire évidemment). Donc, un code 2.7 codé/enregistré en ascii est compatible avec un interpréteur 3.x. Enfin, l'encoding sqlite3 par défaut est l'utf8 (pragma encoding), ce qui lui confère également une compatibilité totale avec du code 2.7 en ascii (et autre création de table ou injection de données interne au programme).

Et pour terminer : *“régler son éditeur sur le bon encoding”* : oui et non : vim va régler l'encoding par défaut sur le “locale” en cours (:set encoding=...) ; mais, si on code en C ou en Python, un :w écrira bien le fichier en ascii à moins d'utiliser des caractères non ascii propre au “locale” en cours (ée ...). Eclipse travaille de la même façon, je peux avoir un “locale” en latin-1 (iso8859-1), coder sous PyDev ou CDT, le résultat sera bien un fichier ascii. Mes 2 kopecs. Merci pour l'article. cdt,



Marco

02/08/2014 at 12:42

J'essai un peu le langage python sur l'ide netbeans, franchement pas convaincu... la doc existe en français ? (mais c pas sûr d'y comprendre d'avantage).

Kel bordel l'encoding, j'arrive jamais à savoir kel encode y faut utiliser mais dans netbeans, tou s'affiche bien (lequel utilise par défaut windows-1252). dans la console y'a pas un accent qui s'affiche correc. jamais vu un langage aussi ambigue et confus... chiotte !



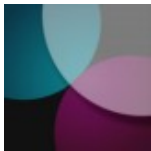
Sam

Post author

02/08/2014 at 16:25

J'ai jamais vu de doc en fr.

Pour ton encoding, il faut que tu relise l'article. Il te faut connaitre l'encoding de ta console, comme pour tous les langages du monde.



Marco

03/08/2014 at 11:04

Bon, ok, j'ai fini par comprendre un peu mieux le sujet et comment Python s'est adapté à la norme Unicode. J'ai surtout compris qu'à force de fouiller un *string* on finit par découvrir qu'il est en fait constitué de *Bytes* lesquels on des possibilités de "petits" ou "gros bouts" selon les types utf-16 BE et LE ?

Cela dit, le comportement du code n'a pas les mêmes répercussions depuis la console de Netbeans ou exécuté par le fichier sur le desktop. Dans l'interpréteur Python, j'ai bien les caractères accentués qd `varText.decode('utf-8')`. La même instruction dans la console de l'IDE lève une erreur:

AttributeError: 'str' object has no attribute 'decode'

Sauf si `VarText = 'chaine'.encode('utf-8')` à été auparavant formulé. Donc, ça roule qd même pas mal mais faut bien comprendre quel type d'objet est restitué dans les manip.



Sam

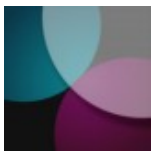
Post author

03/08/2014 at 12:57

Il y a une différence fondamentale entre Python 2 et 3, et selon que tu manipule une chaîne ou des bytes. Et bien entendu, il y a une différence entre l'encoding de ton IDE, celui de la console de l'IDE, et celui de la console de l'OS.

Encore une fois, relis l'article. Il te faut ces informations pour répondre à ta question.

Ce n'est pas spécifique à Python, il faut connaitre l'encoding dans tous les langages du mondes.



Marco

04/08/2014 at 08:09

Ok pour l'encoding, qu'en est-il de la charmap de Python ?

print("\u0100')

UnicodeEncodeError: 'charmap' codec can't encode character '\u0100' in position 0: character maps to

Pourquoi Python ne trouve pas un point de codage dans la table Utf-8 ?



Sam

Post author

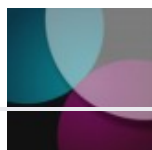
04/08/2014 at 12:49

Ce n'est pas que Python ne trouve pas le point de codage dans la table UTF8, c'est que ce caractère n'existe pas dans l'encodage de sortie. Tu essayes de faire passer un cube dans un trou rond.

Si tu affiche ce caractère dans une console supportant l'UTF8, ça marche parfaitement.

Je me répète donc pour la 3eme fois, relis l'article car tu n'as récupéré aucune des informations que je souligne comme étant indispensables à avoir.

Tu ne m'as jamais donné ton OS. Ni ta version de Python. Ni quelle était ta sortie et son encoding.



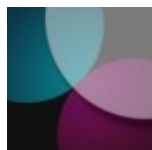
Marco

04/08/2014 at 23:08

Effectivement ça manque d'infos, je suis sur W7-64 et j'utilise Python33.

Apparemment le pluggin Python pour Netbeans développé par le support Oracle comporte un bug. J'ai contacté le support pour avoir des infos et comment résoudre ça, j'attends la réponse.

Je crois devoir chercher un autre IDE pour le dev en Python.



Marco

05/08/2014 at 06:42

Re, j'insiste encore, quitte à devenir un peu lourd, j'ai beau relire l'article, je ne vois nulle part le principal intéressé...

On est bien d'accord que l'encoding sert à transformer une grille de chose d'une matrice sur une autre, au final c'est ce que ça fait, découvrir où sont placés des éléments dans un tableau...

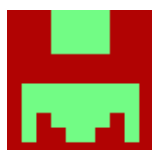
Ici les éléments en question sont des signes typographiques, lesquels plus globalement sont des dessins appelés glyphes.

Nulle part cet article n'explique comment ces glyphes peuvent s'afficher sur un écran ... nulle part ni Unicode ni les grilles UTF-8 n'indique comment dessiner sur un écran ces glyphes.

Je n'ai pas été assez précis sur la nature des erreurs rencontrés certes, mais bien souvent il n'y a simplement rien ou bien juste des carrés qui s'affichent pour remplacer les glyphes. Ça veut dire que l'encoding fonctionne mais l'affichage est incapable de produire le glyphes correspondant.

En changeant la police d'Output par défaut de l'IDE (Arial MS Unicode), les caractères à présent s'affiche correctement.

CQFD. Peut être sur d'autres terminaux cette substitution de caractères est automatique mais en l'occurrence Netbeans avec le pluggin Python ne la produit pas. Au moins ça aura servi à clarifier ce point non négligeable.

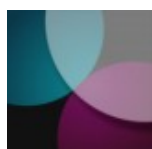


Sam

Post author

05/08/2014 at 12:11

Ah, mais ça n'a rien à voir avec Python ça, c'est juste que tu utilises des polices qui n'ont pas les glyphes en UTF8. Imagines si chaque articles doit répondre à toutes les config, on est pas rendu...



Marco

06/08/2014 at 11:18

Oui, c'est vrai que c'est parallèle au sujet...

Autre question: j'essaie de changer l'encoding de sortie qui semble boulonné à l'objet STDOUT.

En parcourant Stackoverflow il y a une flopée de post sur ce sujet mais pas de réponse ouvrant sur une modif de la config.

Toujours en **Python 3.x** sous W7 64, voilà les données:

```
print(sys.getdefaultencoding()) →UTF-8
```

```
print(locale.getpreferredencoding()) →cp1252
```

Je comprend pourquoi alors que tout est UTF-8, je n'arrive pas à mettre le STDOUT de même. Une solution à été formulé, elle fonctionne et c'est la plus courte observé:

```
utf8stdout = open(1, 'w', encoding='utf-8', closefd=False)
```

```
# fd 1 is stdout
```

```
print(everyfile, file=utf8stdout)
```

Avec ça effectivement il n'y a plus d'erreur de charmap mais bon c'est juste un hack, y'a pas une soluce plus propre ?



Sam

Post author

07/08/2014 at 14:40

Mettre le sdtout en en UTF8 ne veut rien dire. Sdtout est juste un objet qui représente la sortie standard.

Python n'a aucun contrôle sur le périphérique de sortie. Si le périphérique ne supporte pas l'encoding donné, on ne peut pas l'utiliser. C'est aussi simple que ça. On ne peut pas afficher des caractères UTF8 sur une console qui ne supporte pas l'UTF8. Avec aucun langage du monde. Ce n'est pas particulièrement lié à Python.

Vu que je ne sais toujours pas ce que vous voulez faire car vous ne l'avez toujours pas écrits, je n'ai aucun moyen de vous aider. De surcroît, les commentaires de blog sont très mal adaptés à l'aide des gens en ligne.

Mon conseil :

- relisez l'article (bis, ter, etc). Vous avez visiblement raté des choses importantes.
- suivez les conseils des choses à lister pour s'en sortir avec l'encoding. Après autant de message et autant d'appel de ma part à relire l'article, vous n'avez TOUJOURS PAS listé l'encoding de votre sortie ni précisé si votre texte était uniquement issu de strings du code où d'une autre source.
- rédigez votre message de demande d'aide en suivant ce template : <http://sametmax.com/template-de-demande-daide-en-informatique/>. Vraiment, jouer au chat et à la souris pour récupérer des bouts d'indices pour aider quelqu'un qui programme à rédiger une demande d'aide décente, c'est fatigant.
- postez le dit message sur un forum.

Je pense comprendre votre problème, mais vous ne me facilitez pas la tâche pour le résoudre. Je m'arrête donc là.



WhatTheHell

12/08/2014 at 11:27

Merci...

au delà du fait de la qualité de l'article.

Il exprime mon état de nerf face à ce p***** de problème d'encoding entre 2 PC avec 2 versions de windows et 2 versions de python qui essaient de communiquer ensemble. (oui, j'aime me faire du mal..."foooooettez-moi bien fort, maîtresse!!")



Thomas

12/09/2014 at 09:12

Superbe article.

Il m'a beaucoup éclairé sur le sujet.



hengys

27/09/2014 at 12:06

Vraiment merci pour le tuto. Cela fait plus de 3 semaines que je galère avec cette histoire d' encodage. Probleme enfin resolu. Merci vraiment.



Yax

15/10/2014 at 06:57

Une explication complète de A à Z pour éviter la migraine :-)

Merci beaucoup



Yves Masur

15/11/2014 at 16:46

Excellent! Merci pour cette synthèse appréciée, je partage.





Biganon

02/12/2014 at 22:06

Pourquoi avoir préfixé d'un "u" la chaîne dans `msg = u"Page '{}'`
`sauvée\n".format(nom)` alors que tu importé `unicode_literals` ?



Sam

Post author

04/12/2014 at 14:55

La force de l'habitude. Ca n'a aucune impact.



Pwalkour

28/12/2014 at 23:44

Bonsoir, python, c'est tout nouveau pour moi désolé si je dis des bêtises ...

j'vu dans les commentaire qu'une autre personne a rencontré le même que moi, à savoir le problème de noms de fichiers avec des accents lorsque que l'on utilise `walk` pour parcourir une arborescence. Je ne pense pas avoir vu de solution où de suggestion ...

l'un de vous pourrait-il m'aider ?

Avant d'oublier, merci pour cette synthèse très pratique.

A bientôt et bonnes fêtes de fin d'année à tout le monde.

Pwalkour



Sam

Post author

29/12/2014 at 08:36

En python 2.7 :

```
>>> from __future__ import print_function
>>> import os
>>> os.path.walk('/tmp', lambda args, root, files: pri
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
```

Donc les chemins sont de types bytes, et il faut décoder, comme indiqué dans l'article.

```
>>> import os
>>> for root, dirs, files in os.walk('/tmp'):
>>>     print(type(root))
...
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

Donc les chemins sont de types unicode, et il n'y a rien à faire.

Si tu lis l'article, tu sauras exactement quoi faire avec ces informations, il n'y a rien de magique.



enebre

10/01/2015 at 00:20

Bonjour à tous,



les recherches pour un soucis avec python m'ont amené ici, alors je me disais que peut être trouverais-je la solution au problème.



J'utilise depuis quelques temps mps-youtube, petit outils fort sympathique je dois dire.

Mais depuis une mise à niveau rien ne va plus, il me sort une erreur tel que celle exposée ici...

donc j'installe:



```
sudo pip install mps-youtube
```

puis je lance le soft :

```
mpsy
```

j'envoie une recherche,

```
.pantera
```

je lui demande les infos d'une vidéo

```
d2
```

ensuite je demande le téléchargement, c'est là que l'erreur se présente :

```
10
```

mps-youtube a écrit :



```
Problem playing last item: 'ascii' codec can't decode byte 0xc3 in position 13: ordinal not in range(128)
```

Que puis-je faire ?



Sam

Post author

10/01/2015 at 08:07

Ta question n'est pas vraiment lié à l'article car tu ne programmes pas, mais utilise le logiciel, donc le mieux c'est que tu la pose ici : indexerror.net.



Dominique

27/01/2015 at 12:30

Bonjour, j'essaye d'envoyer des séquences d'échappement à une imprimante HP en réseau (python 2.7 sous Windows, 'cp850' pour stdout et 'mbcs' pour le système de fichier).

J'ai essayé de fixer mes problème d'encoding en lisant l'article et fait des gros gros gros efforts (voir code ci-dessous) pour bosser en unicode!

Si je tape le tout en python interactif, cela marche a moitier (mais séquences d'échappement sont ignorées... tout ce qui est éventuellement lisible est imprimer... comme @PJL JOB NAME =)

Le plus étrange, si j'exécute script avec python script.py j'ai l'erreur d'encodage suivante (le script est pourtant sauvé en utf8 par l'éditeur PSPAD).

Traceback (most recent call last):

File "hp-minimal-job.py", line 55, in

```
do_simple_pcl_print_job()
```

File "hp-minimal-job.py", line 45, in do_simple_pcl_print_job

```
print (u".join( l ))#.encode( sys.stdout.encoding )
```

File "C:\Python27\lib\encodings\cp850.py", line 12, in encode

```
return codecs.charmap_encode(input,errors,encoding_map)
```

```
UnicodeEncodeError: 'charmap' codec can't encode character u'\u2013' in position
```

2: character maps to

Qu'est ce que j'ai bien pu raté?





```
#!/usr/bin/env python

— coding: utf8 —

import socket

import sys

HP Network Printer Connexion Point

PRINTER_IP = '192.168.1.206'

PRINTER_PORT = 9100

PJM is used to manage the job and encapsulate a PCL page

PCL is the content to be printer.

#

def pjl_escape( cmd ):

    """ Prepare an escaped command string with cmd as command
    content """

    return u"+chr(27)+cmd"

def pjl_eol():

    """ return the pjl Enf-of-Line composed of space+cr+lf """

    return u' '+chr(13)+chr(10)

def prepare_pjl_job( jobname ):

    """ initiable a PJL job + switch to PCL language """

    l = [] # prepare list of string

    l.append( pjl_escape(u'%-12345x')+str(u'@PJL JOB NAME =
    "+jobname+str(u"")+pjl_eol() )

    l.append( u'@PJL ENTER LANGUAGE = PCL'+pjl_eol() )

    return u".join(l)

def close_pjl_job():

    return pjl_escape(u'%-12345x')+str(u' @PJL
    EOF')+pjl_eol()+pjl_escape(u'%-12345x')

def do_simple_pcl_print_job():

    s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

    s.connect( (PRINTER_IP, PRINTER_PORT) )

    l = []

    l.append( prepare_pjl_job(u"invoice x customer y") )

    l.append( u'Essai d impression é@ testing' ) # PCL command here

    l.append( close_pjl_job() )

    A imprimer

    print (u".join( l ))#.encode( sys.stdout.encoding )

    for c in (u".join( l ))[0:-1]:

        print( ('%03i ' % ord(c)).encode( sys.stdout.encoding) )

    Envoyer à l'imprimante

    s.sendall( (u".join(l)).encode( 'cp850' ) ) # Ne pas envoyer le
    dernier caractère qui est un NULL

    s.close()

if name == 'main':

    do_simple_pcl_print_job()
```



Sam Post author

27/01/2015 at 13:53
Bonjour dominique,
Pose ta question ici : indexerror.net





dacrovinunghi

14/02/2015 at 21:24

putain merci !



TurtleCrazy

06/03/2015 at 14:27

Il faudrait toutefois clarifier ce que demande python3, parce que dire que c'est de l' **Unicode** par défaut n'est pas suffisant.

Unicode est un standard et une norme. C'est tout.

UTF décrit une "transformation" qui répond au standard Unicode et UTF-8 en est le pendant codé sur 8bits.

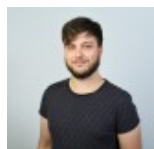
Par contre, préfixer les chaines avec un "u" correspond en général à déclarer une chaine de caractères codée en UTF-8.

Donc quel est l'encodage par défaut d'un compilateur python3 ? Sachant que dans le monde Windows, le terme 'Unicode' est mal employé pour désigner de l'UTF-16(*) alors que dans le monde Unix, il est tout aussi mal employé pour désigner de l' UTF-8 ...

D'autant que le compilateur ou l'interpréteur peut lire un fichier dans un encodage et en utiliser un autre en interne (un pratique courante est de lire de l'utf-8 ou du latin-1 par défaut – voire exclusivement- et de stocker les chaines en UTF-16)

Ce qui d'ailleurs enfonce le clou pour affirmer que l'encodage *doit* être précisé et ne doit jamais être "supposé" par défaut...

(*): pire encore, il y a deux UTF-16 (selon l' endianess) et je ne sais pas lequel est utilisé en interne sous Windows.



Julian

13/03/2015 at 23:44

Salut et merci pour l'article!

J'ai relevé une mini erreur pour la déclaration d'encoding du fichier : `# — coding: utf8 —`

Après essai python2.7 retourne une erreur car c'est : `# — coding: utf-8 —`

C'est tout, simple correction de mini coquille :)

Good night



Sam

Post author

14/03/2015 at 08:57

Hum, normalement ce sont des alias, et les deux marchent sur ma machine. Une spécificité windows ? A creuser.



Fred

08/05/2015 at 08:54

Oh *;:@\#% de *;:@\#% ! L'article que je cherchais depuis 2~3 ans !

Une nouvelle ère s'ouvre pour moi désormais.

Merci

"Je sais une chose : c'est que j'étais aveugle, maintenant je vois"
Jean IX.25



Drigo

22/06/2015 at 19:32

J'adore le thon employé sur ce site (c'est pas une faute au niveau cul ;-)))



Dans le genre branlette, je suis tombé sur (je parle d’accent en l’occurrence)



sur la différence entre les accents composés et les accents décomposés (même moi je n’y ai pas cru)

Genre un è et un ë voire un è et un è, ça saute pas à l’œil mais il y’a plusieurs manières de le faire, vu le niveau

des agitateurs je me dis, et vu l’article que je viens de lire (c’est bon les gars, respect :-))

Si ça continue je file élever des chèvres dans le Larzac ;-)

(et comme disais @Coluche il n’y a pas de meeeehhhhh)



Gwenmael

13/08/2015 at 10:15

Bonjour,

Pouvez vous m’expliquer pourquoi j’obteint systématiquement une erreur ? faut-il que j’importe une bibliothèque particulière ?

IDLE 2.6.5 ===== No Subprocess =====

```
coding: utf8
une_chaine = 'Chaîne'
type(une_chaine)
une_chaine = une_chaine.decode('utf8')
Traceback (most recent call last):
File "<pyshell#4>", line 1, in
une_chaine = une_chaine.decode('utf8')
File
"C:\ESRI\Python26\ArcGIS10.0\lib\encodings\
utf_8.py", line 16, in decode
return codecs.utf_8_decode(input, errors,
True)
UnicodeDecodeError: 'utf8' codec can't
decode bytes in position 3-5: invalid data
```

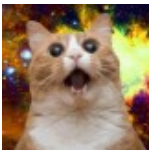


Max

13/08/2015 at 18:11

Salut,

Tu devrais poser ta question sur <http://indexerror.net>, tu auras plus de chance d’avoir une réponse et ça pourra servir à d’autres.



Eildosa

15/09/2015 at 11:34

Bon je croyais avoir tous compris mais je tombe sur cet article.

Pourquoi ASCII est dans la liste des encodage alors que c’est supposé etre un jeu de caractère? (qui pourrais etre encodé en utf-8 par exemple)



Sam

Post author

16/09/2015 at 06:19

ASCII et UTF8 sont tous les deux des jeux de caractères (character sets). L’encodage (encoding) est le processus qui transforme un jeu de caractères en un autre. Ppour cette raison, on fait , comme dans cet article, souvent l’abus de langage en parlant d”encoding ASCII” ou “encoding UTF8” alors qu’on devrait dire “charset ASCII” et “charset UTF8”.



Comme les jeux de caractères ne sont pas équivalents, l'encodage en Python se fait par une forme intermédiaire, vers laquelle on `decode()` et depuis laquelle on `encode()`, ce qui permet de facilement identifier quand le processus échoue : au niveau du charset de départ ou de celui d'arrivée. Cela permet aussi d'isoler le texte comme un type à part en Python 3 : `str` est considéré comme du texte pur, tandis que tout texte encodé est sous forme de `bytes`.

En vérité, `str` est implémenté en utilisant `utf8`. Mais on a pas besoin de le savoir, et cela peut changer. C'est juste une API pour donner :

- une abstraction du texte. Le jour où le texte évolue, on garde la même API et on peut changer l'implémentation.
- de bonnes habitudes sur le mécanisme d'encodage/décodage.
- un débogage plus facile avec des erreurs très précises et des choix explicites sur leurs traitements.



Olivier Pons

22/11/2015 at 23:00

Si jamais on met

`coding: utf8`

alors sous Django, `makemessages` ne fonctionne pas, il ne reconnaît pas `"utf8"`

Si on met :

`coding: utf-8`

alors sous Django, `makemessages` fonctionne, et python fonctionne tout aussi bien. Voilà ma petite contribution...



Sam

Post author

24/11/2015 at 14:19

Bon à savoir.



Jérôme

13/12/2015 at 22:53

Salut j'ai testé un truc à la con:

le script contient:

```
# encoding: utf8
from __future__ import unicode_literals, print_function

for v in sys.argv:
    v2 = v.decode(locale.getdefaultlocale()[1])
    print(v2, type(v2), locale.getdefaultlocale()[1])
```

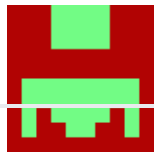
J'ai lancé le script en ligne de commande exécuter `cmd /k`

contenu de la fenêtre cmd:

```
E:\temp>chcp
Page de codes active : 850

E:\temp>python pytest.py "à@^``ùéôö"
pytest.py  cp1252
à@^``ùéôö  cp1252
```

Donc le `stdin` est réinterprété avec la l'encodage local par défaut et la commande `print` converti automatiquement l'encodage du script vers celui du `stdout`



Sam

Post author

14/12/2015 at 20:55

Ca serait intéressant de voir si ce comportement est :

- commun à tous les OS et les terminaux;
- identique en 2.7 et 3



Jérôme

14/12/2015 at 21:19

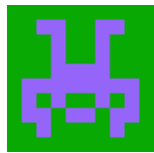
En effet avec une machine virtuelle surement mais ce qui est bon a savoir c'est que si l'on fait un chcp avec un autre encodage python est perdu



Jerome AMON

16/12/2015 at 16:38

Merci pour ce post. Très riche. Je viens de m'en servir pour régler le problème d'encoding dans mon application personnel de Chat en Python 2.7. Excellent ! Je migre bientôt vers la 3.x



Jean-Baptiste

11/01/2016 at 15:21

Bonjour,

Merci pour cet article. Cependant vous dites que :

Il existe un encoding qui essaye des regrouper toutes les langues du monde, et il s'appelle unicode.

Or Unicode n'est pas un encoding, c'est un jeu de caractères (un character set), un concept théorique détaché de la problématique du stockage sur disque dur si je comprends bien l'article de Joël Spolsky à ce sujet en français ou en anglais. Unicode associe des caractères à des "points de code" en hexadécimal, sans se soucier du stockage. Par ex. le point de code Unicode de "A" est U+0041. D'après Spolsky, "UTF-8 est un [autre] système pour stocker en mémoire vos chaînes de points de code Unicode" et comme il le dit plus loin dans son article "on encode de l'unicode" [en UTF-8, en UTF-7]. Mais on n'encode pas une chaîne de caractères en Unicode, ça ne veut rien dire.

Ce qui permet de comprendre pourquoi vous dites plus loin que :

Ensuite, il existe deux types de chaînes de caractères en Python :

...

*La chaîne de caractères **décodée**: type 'unicode' en Python 2.7, et 'str' en python 3 (sic).*

Sinon, si Unicode est un encoding et si une chaîne est en Unicode, pourquoi dirait-on qu'elle est "décodée". On dirait plutôt qu'elle est "encodée" en Unicode. Ce qui n'est pas le cas.



Jérôme

03/03/2016 at 17:32

@Jean-Baptiste

En effet c'est un abus de langage de dire ce c'est un encodage mais...

Le fait d'encoder ou décoder que l'on soit en unicode ou non. Tout dépend de quel charset ou codepage défini comme natif sur ton système ou ton langage... donc il y a décodage pour faire correspondre vers ton système et encodage pour les sorties différentes du codepage système employé.

@ast





Oui est non. le codepage d'un fichier texte n'est pas forcément reconnu si le codepage n'est pas présent dans celui-ci. Il y a aussi des application qui ne prennent que le codepage du système pour l'ouverture et qui enregistre dans ce même codepage si le fichier n'en avait pas un de renseigné au départ. C'est dans ces cas que l'on se retrouve en base de données avec des caractères des plus bizarres...

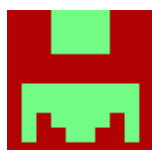
En clair : amuses toi à ouvrir un csv avec Excel en Utf-8 avec un Windows et tu verra qu'il te fera bien de la merde. le codepage par défaut est le CP-1252 (en France) donc tous les caractères non ascii... ben... tu peux toujours te les mettre ou je pense. Certains fichiers n'ont pas cette info car elle se trouve dans une en-tête du format de fichier, dans des métadonnées, en balise dans le contenu du texte (cas du xml et de ses dérivées comme xhtml) Certaines applications ne s'en préoccupe pas donc c'est à toi de savoir comment fonctionne ton application ou ton langage et de t'adapter aux besoins...

On peut aller encore plus loin... Ton windows FR travaille en CP-1252 et tu travailles sur un soft anglais de traitement de données GPS qui lui est codé pour travailler avec un charset différent et des paramètre de séparateur décimal en . et non en virgule.

Résultat : Les données stockées dans le fichier avec des vigules ne sont pas reconnu comme étant du numérique. Le texte à l'affichage est pas propre car il ne reconnait pas le charset...

Du coup, lors de l'ouverture tu dois proposer: 1) choisir l'encodage du fichier d'entrer, 2) choisir le type de séparateur. Rien n'est vraiment automatique.

En clair, il faut connaitre si le charset est stockées dans le fichier ou dans le jeu-de-données ou dans certains cas dans une readme.txt ou encore (de mémoire du fournisseur...). Dans certains cas c'est un fichier situé au même endroit avec une extension *.cpg qui donne cette info d'où la notion de jeu de données. Encore faut-il que l'application qui ouvre ce fichier soit capable d'identifier son existence...

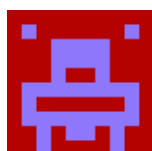


Sam

Post author

11/01/2016 at 17:53

Oui c'est un abus de langage.

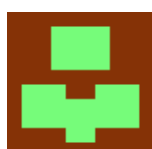


ast

03/03/2016 at 16:42

Quand on ouvre un fichier texte avec n'importe quel éditeur, jamais on ne nous demande quel est l'encodage, et heureusement lol. L'éditeur se débrouille.

Comment ? Avec une fonction de détection comme chardet peut être ?

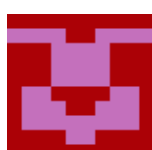


foxmask

11/03/2016 at 21:44

Toujours à l'affut de saloperies qui trainent, je tombe sur un pb de ce genre et me dit qu'un de flux RSS que je grab est mal goalé alors je retourne voir FeedParser et tombe sur Universal Encoding Detector (aka chardet) qui semble être de FeedParser et non de mozilla, à moins que mr FeedParser ne soit un Mozillian.

Bon en tout cas me v'la reparti check les flux



turboiii

20/04/2016 at 07:51

Bonjour



comme prédit par l’oracle « Sam » j’ai fini par tomber dans le cas où mon programme buggue à cause de l’encodage.



Fatal error !!!!!!! blocage complet je ne comprends plus
.....doute.....

et puis je reviens sur le site. Retour aux sources.....divines

Ah oui, ben oui.

Bêtement j’applique les 5 règles.

Que se passe t-il ce con de DrPython ne veut même plus de mon fichier source. Ouf j’ai une sauvegarde....

Je repars de zero avec un fichier qui trempe dans gtk3 ou gtk+ comme tu veux.

Et là catastrophe

<https://python-gtk-3-tutorial.readthedocs.org/en/latest/unicode.html>

==> gtk semble ne pas respecter tout les critères pas gloop du coup j’investigue et

point 1

coding: utf-8 en première ligne avec mon éditeur en utf-8 est ok

point 2

avec gtk

from **future** import unicode_literals n’est pas applicable sinon sa plante

par conséquent je suis contraint d’écrire les chaines comme suit

text = u"Fichier sélectionné"

surprise refus !!

je reviens sur

text = "Fichier sélectionné"

je vérifie les types je ne fais plus confiance à rien

plus de bug, RUN

en sortie de stout je sors des caractères non ascii???bigre

vite vite l’option ‘replace’

ah Fichier s❖lectionn❖ c’est beaucoup mieux :-)

je retourne à un vieux projet

pour déterrer ça

sys.stdout = codecs.getwriter("utf-8")(sys.stdout)

ca à l’air de fonctionner au détail près que je suis obligé d’encoder en latin pour une sortie propre à la console

le code employé de la salle de torture

```
❏  
[  
coding: utf-8  
  
toutes les chaines sont en unicode (même les docstrings)  
  
from future import unicode_literals  
  
filechooserdialog.py  
  
inspiré de test_filechooserdialog_1.py  
  
import sys  
  
import gi  
  
import codecs  
  
gi.require_version('Gtk', '3.0')  
  
from gi.repository import Gtk  
  
class FileChooserDialog():  
  
    """ choose file widget with Gtk.FileChooserDialog"""  
  
    def init(self,debug= False):  
  
#Stores your path
```



```
self.path = None

self.debug = debug

self.dia = Gtk.FileChooserDialog("Please choose a file", None,
Gtk.FileChooserAction.OPEN,

(Gtk.STOCK_CANCEL, Gtk.ResponseType.CANCEL,

Gtk.STOCK_OPEN, Gtk.ResponseType.OK))

self.add_filters(self.dia)

self.dia.set_select_multiple(True)

def run(self):
    self.reponse = self.dia.run()
    if self.reponse == Gtk.ResponseType.OK:
        if self.debug:
            print("Open clicked")
            print("File selected: " + self.dia.get_filenames
())

            for e in self.dia.get_filenames():
                print e
            self.path = self.dia.get_filenames()
    elif self.reponse == Gtk.ResponseType.CANCEL:
        if self.debug:print("Cancel clicked")
        self.dia.destroy()
        return self.reponse

def get_files(self):
    ''' return list of selected file '''
    return self.path

def add_filters(self,dialog):
    ''' example could be filtered file '''
    filter_text = Gtk.FileFilter()
    filter_text.set_name("Text files")
    filter_text.add_mime_type("text/plain")
    dialog.add_filter(filter_text)

    filter_py = Gtk.FileFilter()
    filter_py.set_name("Python files")
    filter_py.add_mime_type("text/x-python")
    dialog.add_filter(filter_py)

    filter_any = Gtk.FileFilter()
    filter_any.set_name("Any files")
    filter_any.add_pattern("*")
    dialog.add_filter(filter_any)

if name == "main":

exemple = FileChooserDialog()

response = exemple.run()

if response == Gtk.ResponseType.OK:

text = "Fichier sélectionné"

print type(text) # donne normal

text = unicode(text,'utf-8', 'replace')

print type(text) # donne normal

text = text.encode('utf8', 'replace')

print(text) # sortie avec les caratères non ascii remplacé par
des ? normal

#####

print '#####'

sys.stdout = codecs.getwriter("utf-8")(sys.stdout)

text = "Fichier sélectionné"

text = unicode(text,'utf8', 'replace')

print(text)

text = "Fichier sélectionné"
```




```
text = unicode(text,'ISO-8859-1', 'replace')

print(text) # va sortir propre sur la console alors que la
console serait codé en utf-8 ????
```

```
for e in exemple.get_files():

print e

}
```

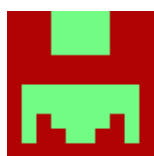
Ton avis sur cette affaire ?

Sans remettre en cause gtk bien sur

MA QUESTION : avant de continuer à redresser les divers codes de mon projet suis-je sur la bonne voie et à l'abri de nouvelles surprises ???

désolé j'ai utilisé les tag code et /code mais je perds l'indentation

sinon merci Sam pour cette prose fournie sur python. C'est inspirant en tout cas. Humblement il y a des articles qui me dépassent d'autres qui me permettent de progresser et ça c'est génial

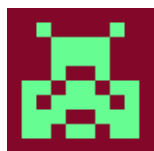


Sam

Post author

21/04/2016 at 06:37

=> indexerror.net



tocard 2016 wiiiz

24/07/2016 at 23:03

petite source d'inspiration:

Ce MONSIEUR est un core dev de cython et a écrit un manifest sur l'unicode en python:

https://github.com/haypo/unicode_book/blob/master/good_practices.rst



Pomme

07/10/2016 at 08:31

En pratique sur un texte en français, le surcoût de l'utf-8 par rapport à l'ignoble ascii est de moins de 5 % : en effet les caractères latins non accentués restent encodés sur 1 octet (comme l'ascii) et seuls les caractères accentués le sont sur deux octets... Donc à mon humble avis l'optimisation de la mémoire n'est pas une bonne excuse pour utiliser l'ascii. En fait si vous utilisez l'ascii, on va vous taper. fort.



Tidus

23/10/2016 at 10:14

Bonjour,

Tout d'abord un grand merci pour ces explications.

Toutefois j'ai un gros problème d'encodage et je ne comprends pas tout ce qui est indiqué ci-dessous (faut dire que je suis un gros débutant par rapport à vous autres). Voici mon problème:

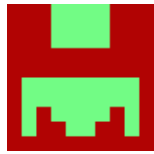
File "C:\Python27\lib\encodings\cp1252.py", line 15, in decode
return codecs.charmap_decode(input,errors,decoding_table)

UnicodeDecodeError: 'charmap' codec can't decode byte 0x81 in position 60: character maps to

Cela fait presque 24h que j'essaye de régler le problème mais je n'y arrive pas. Je comprends qu'il y a un problème d'unicode et d'ansi, mais malgré tout ce que j'ai pu voir et faire je n'y arrive pas.

(Pour plus de précision j'essaye pour ceux qui connaissent d'installer "wrye bash" sur le jeu Oblivion)



**Sam**

Post author

23/10/2016 at 21:18

Bonjour Tidus. Va sur indexerror.net, on pourra mieux t'aider là bas.

**Gio***10/02/2017 at 14:31*

Salut,

article bien complet et utile ... sauf que je n'arrive pas à m'en sortir :

je suis sur une VM Debian , je travaille avec eclipse.

Je liste une directory qui contient un fichier "scène.txt"

Le résultat de `os.listdir(dir)` est "sc?ne.txt" et c'est bien un point d'interrogation code ascii 63.

Un `sys.getfilesystemencoding()` donne ANSI_X3.4-1968 (d'où ça sort ???)

`echo $LANG` donne en_US.utf8

Que faire ?

muchas gracias.

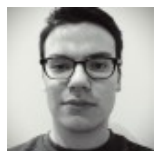
Sergio

**Sam**

Post author

10/02/2017 at 16:03

Va plutôt poster ta question sur indexerror.net, on y répond là bas. Ici c'est pas pratique.

**Quentin***17/03/2017 at 14:32*

Merci tellement pour cet article !

On est en 2017, mais c'est encore aujourd'hui le seul article français (voire du net), à expliquer aussi bien l'encoding. Surtout pour un pauvre mec comme moi qui a appris à coder seul (sans background académique) et qui galérait comme un con avec ma pauvre BDD qui me donnait des caractères chelous quand j'entrais du texte français...

Sam, je t'aime <3

**AlainFELER***19/03/2017 at 14:49*

Depuis le temps que je reviens à cet article... faute d'orthographe dans le titre : une fois pour toute → une fois pour touteS !

En tout cas merci(s).

**AlainFELER***19/03/2017 at 15:10*

... je ne comprend pas comment (en Python 2.7) combiner vos infos et l'usage du module csv natif de Python.

Je veux modifier des csv en Windows-1252 ou UTF-8 (je reçois l'info en paramètre).

Par ex. j'ai en entrée des colonnes du genre "Prénom Nom" et des valeurs du genre 'Rémi Dufrêne' et je veux en sortie 'PRENOM','NOM' et 'Rémi','Dufrêne'



La doc Python <https://docs.python.org/2/library/csv.html> dit que le module csv ne gère pas l'unicode et explique des trucs en 13.1.5 que je ne comprends pas.



Sam

Post author

19/03/2017 at 15:58

C'est exactement ça: csv en 2.7 ne gère pas l'unicode. Il existe un backport du module csv de python 3 pour y pallier : <https://pypi.python.org/pypi/unicodcsv/0.14.1>



vchalmel

29/06/2017 at 08:49

Bonjour, et merci pour cet article de référence !

J'ai un comportement curieux d'un ETL incluant des scripts python



-- coding: utf-8 --

```
print(type(unicode("Début Logging")))
```

```
print(type(u"Début logging"))
```

renvoient une erreur 'utf8' codec can't decode byte 0xe9

Et effectivement si je fais un print d'un `chardet.detect` de la chaîne de caractère (non unicode) dans ce logiciel il retourne 'windows-1252' (ou Latin-2 selon les accents que j'utilise, comme tu le dis bien la détection n'est pas une science exacte), du coup j'ai l'impression que python en lisant `u"Debut Logging"` tente de faire un `"Début Logging".decode('utf-8')` alors que la chaîne d'entrée est resté dans un autre encodage.

Est-il possible que ma déclaration `# coding : utf8` induise en erreur l'interpréteur python ? J'avais l'impression que ce n'était qu'une consigne pour les éditeurs de texte...

Confirmez vous ? Une solution ?



Sam

Post author

29/06/2017 at 12:57

C'est une spécificité de Python : si l'encoding du fichier est différent de l'encoding par défaut du langage, il faut le déclarer sinon le programme plantera à la première conversion.



pouet

07/06/2018 at 15:53

Salut. Je déterre le thread (c'est moche mais tant pis) pour plusieurs raisons.

La première, c'est pour remercier. Ces dernières années j'ai relu cet article un certain et il m'a beaucoup aidé à comprendre comment fonctionne l'encoding en Python, qui soyons clair, est passablement merdique... Donc voilà, merci beaucoup pour ces infos précieuses.

Deuxièmement, pour émettre des réserves sur un principe énoncé ici et que j'ai longtemps essayé d'appliquer sans succès, à savoir préfixer toutes ses chaînes avec `u"machin"` pour que toutes les chaînes aient le type unicode. De mon expérience, faire ça débouche presque obligatoirement sur de nouveaux plantages d'encoding. Pour deux raisons, mais qui au final reviennent à la même :

1) le type `str` est différent du type `unicode` et par conséquent, une fonction qui attend un type `str` peut avoir un comportement différent si on lui transmet une variable de type `unicode` voire (et c'est le cas



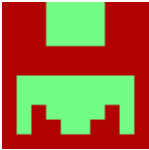
la plupart du temps) planter, souvent pour la raison en 2)

2) la fonction `str()` de python (en version 2.7 du moins) est de la merde et plantera sur toute variable de type unicode contenant un caractère non-ascii, bien que d’après sa doc officielle “return a string containing a nicely printable representation of an object.”. Peut importe donc que tu te sois fais chier à gentiment décoder ta chaîne avec le bon codec, python s’en branle et essaie de la réencoder en ascii à l’appel de `str()`...

Or, en revanche, l’appel de `str()` sur une chaîne encodée (`str`) contenant des caractères non-ascii ne plante pas. C’est un comportement qu’on peut critiquer, mais on peut aussi le voir autrement en estimant que python délègue simplement la responsabilité d’interpréter les caractères au terminal de sortie, Si terminal de sortie il y a. Et justement, dans de nombreux cas, dans le cadre d’utilisation de la chaîne au sein du programme python, on utilise de nombreuses chaînes “transitoires” qui n’ont pas vocation à être imprimées (par python du moins) : clés de dictionnaire, requêtes SQL, etc. Il n’y a donc strictement aucun intérêt à les décoder.

Pour en revenir au comportement de `str()`, pour moi c’est une énorme aberration, surtout que par ailleurs, on a le même problème dans l’autre sens : un appel de `unicode()` sur une chaîne encodée (`str` donc) contenant des caractères non-ascii va planter. La conséquence misérable, c’est qu’il n’existe aucune fonction universelle en python permettant d’extraire la valeur imprimable d’une variable quel que soit son type, alors que c’est ce que `str()` devrait faire ! Dès qu’on construit une fonction de type `logger()` par exemple qui va faire du `str()` sur toutes les variables qu’on lui envoie (peut importe leur type), on va être obligé de vérifier leur type avant JUSTE pour gérer le cas des chaînes décodées, ce qui soyons clair, est complètement débile....

Je suis ouvert bien sûr aux remarques et critiques (si vous avez eu le courage de lire mon pavé :P)



Sam Post author

07/06/2018 at 18:02

Le comportement de `str()` est aberrant parce qu’il ne gère pas du texte, mais des octets bruts représentant du texte.

Quand on a voulu faire le split octets / texte en Python 2, on s’est hâté à un problème: on avait déjà donné le nom `str` à quelque chose qui n’était PAS du texte selon la nouvelle API.

Donc a on fait `str` / unicode, et `str` est le nom resté pour représenter ... ce qui n’est pas du texte.

Du coup tout le monde continue de se gourer en Python 2, voulant manipuler du texte, mais manipulant des bytes sans comprendre la plupart du temps.

En Python 3, on a corrigé ça, et `str()` représente / converti bien du texte, tandis que `bytes()` represente / converti bien des bytes.

Le comportement de Python est donc logique, et même sain, mais pour des raisons de compatibilité, on ne peut pas avoir une sémantique clair en Python 2. Il est en effet, hors de question de casser python 2.



le Fatumbi

03/07/2018 at 09:45

ah bhen super) /e truc à lire avant de tenter d’ouvrir le moindre txt en python 2.7

merci merci :-)

sauf que la video est bloquée pour des raisons de droit d’auteur, mais le reste ça va)





Sam Post author
03/07/2018 at 09:48
Ok, c'est corrigé.



christophe
11/08/2018 at 14:34
Eh bien on peut dire que ça c'est un post aussi long qu'utile !!
Je me bat depuis deux jours avec un module qui grace à tes lumières m'ont permis de comprendre qu'il ne supporte pas l'unicode (img2pdf)
Mille merci pour m'avoir éclairé sur ces foutus problèmes de tables, j'ai enfin pu corriger ce qui n'allait pas
Merci beaucoup !

Comments are closed.

Des questions Python sans rapport avec l'article ?
Posez-les sur [IndexError](#).

Post navigation

← Deux conneries à la seconde

Créer un snippet pour Sublime Text →

Hors du blog

- › [Page de contact](#)
- › [@sam_et_max](#) sur twitter
- › [Nos tweets en RSS](#)
- › [Fork me, I'm famous](#) (github)

Tous les textes de ce blog, sauf
signalement contraire, sont sous
licence creative common 3.0 unported.

Toi aussi, trouve
un article obsolète
sur notre blog

- › [September 2019](#) (1)
- › [June 2019](#) (1)
- › [February 2019](#) (1)
- › [January 2019](#) (3)
- › [December 2018](#) (1)
- › [September 2018](#) (2)
- › [August 2018](#) (1)
- › [July 2018](#) (3)
- › [June 2018](#) (4)
- › [May 2018](#) (2)
- › [March 2018](#) (2)
- › [February 2018](#) (1)
- › [December 2017](#) (4)
- › [November 2017](#) (5)
- › [October 2017](#) (2)
- › [September 2017](#) (1)
- › [July 2017](#) (5)
- › [June 2017](#) (3)
- › [May 2017](#) (3)
- › [April 2017](#) (2)
- › [March 2017](#) (3)
- › [February 2017](#) (5)
- › [January 2017](#) (5)
- › [October 2016](#) (2)
- › [September 2016](#) (5)
- › [August 2016](#) (2)
- › [March 2016](#) (8)
- › [February 2016](#) (9)
- › [January 2016](#) (16)
- › [December 2015](#) (6)
- › [November 2015](#) (7)
- › [October 2015](#) (2)
- › [September 2015](#) (6)



- › August 2015 (10)
- › July 2015 (18)
- › June 2015 (13)
- › May 2015 (12)
- › April 2015 (13)
- › March 2015 (8)
- › February 2015 (4)
- › January 2015 (21)
- › December 2014 (24)
- › November 2014 (6)
- › October 2014 (18)
- › September 2014 (10)
- › August 2014 (5)
- › July 2014 (11)
- › June 2014 (25)
- › May 2014 (14)
- › April 2014 (10)
- › March 2014 (23)
- › February 2014 (27)
- › January 2014 (17)
- › December 2013 (27)
- › November 2013 (25)
- › October 2013 (27)
- › September 2013 (1)
- › August 2013 (31)
- › July 2013 (32)
- › June 2013 (24)
- › May 2013 (30)
- › April 2013 (31)
- › March 2013 (33)
- › February 2013 (29)
- › January 2013 (34)
- › December 2012 (32)
- › November 2012 (36)
- › October 2012 (35)
- › September 2012 (36)
- › August 2012 (37)
- › July 2012 (33)
- › June 2012 (24)
- › May 2012 (31)
- › April 2012 (20)
- › March 2012 (10)
- › February 2012 (14)

