Использование Unicode в Python

Юрий Юревич
http://gorod-omsk.ru/blog/pythy/

Конференция по Ruby и Python.
Омск, 10 февраля 2007

Я часто встречаю непонимание что такое Unicode и как вообще Python с ним работает. Попытаюсь это исправить.

1 Что такое Юникод

Немного предыстории. Изначально компьютеры использовали только один язык - английский. И весь алфавит, все используемые символы уместились в 7 бит, т.е. 128 символ-мест. Это собственно и есть ASCII. Из 8 бит, занимаемых типом char, 1 бит был "запасным". Однако когда компьютеры получили распространение за пределами США, возникла проблема размещения национальных символов. С Западной Европой еще было худо-бедно, там хоть бОльшая часть символов пересекалась с ASCII, а вот с Восточной Европой была вообще беда. В том числе и с русским языком. К текущему моменту в живых остались кодировки cp866 (DOS) с псевдографикой, cp1251, она же windows, koi8-r с интересным дизайном (когда у koi8 "отрезают"старший 8-ой бит, то текст становится транслитом) но без знака №.

Unicode - это стандарт кодирования. Символы являются неким абстрактным понятием, кодируются так называемыми кодовыми точками. В отличии от кодировок, Unicode не описывает конкретное представление символов.

Понятия

Кодировки
Кодировка — это таблица соответствий между символами и их машинными представлениями.
Буква «а»:

- windows-1251 — `E0h`
- cp866 — `A0h`
- koi8-r — `C1h`
- utf-8 — `D0h B0h`

Юникод (unicode)
Юникод — стандарт кодирования. Не говорит о конкретном представлении символа.
Буква «а»:

- unicode — кодовая точка `U+0430`

2 Типы строк в Python

Теперь перейдем к Python. В нем существует два типа строк - обычная 8-битная, и юникод-строка. Внешне юникод строка отличается тем, что перед кавычками идет буква ‘u’. По внешним признакам они более ни чем не отличаются. Другое дело, что 8-битная строка хранится в виде конкретной последовательности байтов, а юникод-строка в виде кодовых точек.

Типы строк в Python

Обычная строка

```
>>> regular_string = 'обычная строка'
>>> type(regular_string)
<type 'str'>
>>> 'a'
'\xd0\xb0'
```

Юникод-строка

```
>>> unicode_string = u'юникод-строка'
>>> type(unicode_string)
<type 'unicode'>
>>> u'a'
u'\u0430'
```

3 Преобразования между типами строк

При преобразовании между этими типами, достаточно помнить, что юникод более общий тип данных. И чтобы преобразовать к нему, нужно знать исходную строку и в какой кодировке она представлена. Т.е. **декодировать** строку.

В случае обратного перехода, из юникода в строку. Опять же, достаточно помнить, что юникод - более общий тип данных, и чтобы получить строку в определенной кодировке, нужно **закодировать** юникод.

Преобразования между типами строк

Строка → юникод

```
>>> regular_string = 'обычная строка'
>>> type(regular_string)
<type 'str'>
>>> unicode_string = regular_string.decode('utf-8')
>>> type(unicode_string)
<type 'unicode'>
```

Юникод → строка

```
>>> unicode_string = u'юникод-строка'
>>> type(unicode_string)
<type 'unicode'>
>>> regular_string = unicode_string.encode('utf-8')
>>> type(regular_string)
<type 'str'>
```

4 Юникод != utf-8

Вот дошла очередь и до UTF-8. Действительно, Платоновские идеи в виде кодовых точек хороши, но как их хранить? Один из способов хранение (т.е. кодировки) и есть UTF-8. С точки зрения англоязычных программистов, было расточительно под каждый символ отдавать два байта, да и обратной совместимости с ASCII хотелось. Так что решили делать так: для кодов меньше 128 - один байт. Для тех что больше - два и более. На практике это выглядит так:

Юникод != UTF-8

Юникод

```
>>> unicode_string = u'юникод строка'
>>> print unicode_string[:3]  # -> юни
юни
>>> len(unicode_string)       # -> 13
13
```

UTF-8

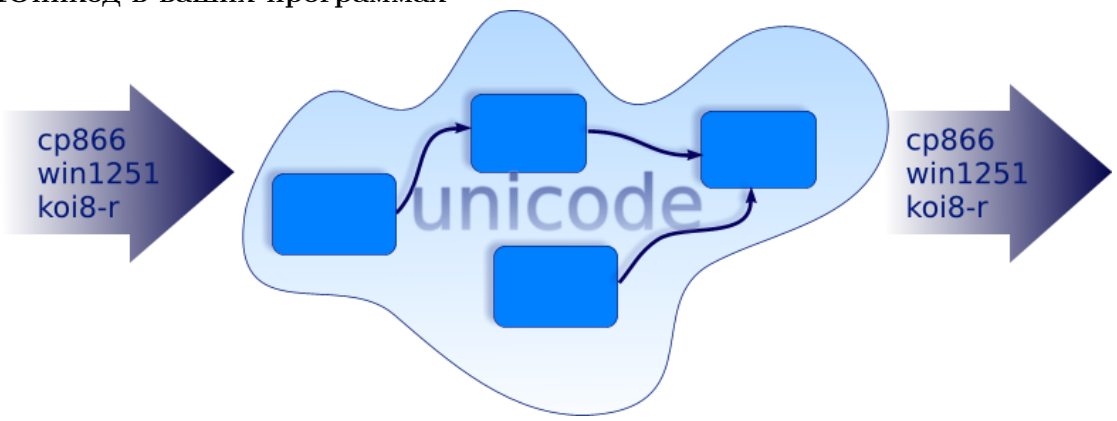
```
>>> utf8_string = 'utf8 строка'
>>> print utf8_string[:3]   # -> utf
utf
>>> print utf8_string[-3:]  # -> ока?
?a  #  WTF?
>>> len(utf8_string)        # -> 11?
17  #  WTF?
```

5 Юникод в ваших программах

Чтобы избежать проблем, связанных с нюансами кодировок, следует для внутреннего представления выбирать юникод. Входные и выходные данные могут быть как юникодом, так и обычными строками в произвольной кодировке. Но использование юникода позволит сразу решить ворох проблем: различные кодировки на различных платформах, единообразное поведение для любых входных данных, потери при перекодировках и др.

В итоге, совет звучит так: явно перекодируйте данные на входе/выходе и используйте юникод для внутреннего представления.

Юникод в ваших программах



6 Частые ошибки

Далее, я расскажу о типичных ошибках при работе с юникод

Первая ошибка - неявное перекодирования в юникод. В этом случае в качестве исходной кодировки берется кодировка, определенная в site.py - глобальном конфигурационном файле Python. По умолчанию это ascii. Чаще всего эта ошибка встречается у англоязычных разработчиков, для большинства которых юникод == ascii == 8 бит. Симптоматику вы видите - это исключение UnicodeDecodeError. Видя такие исключения, да еще и не в своем коде, первая мысль русскоязычного разработчика - исправить в site.py кодировку. Это неправильная мысль. Правильное же решение - указать разработчику на ошибку и использовать явную перекодировку.

Первый вариант (с unicode) универсальней - первым аргументом может быть как строка, так и юникод. Но вспомним дзен Python - явное лучше неявного. На мой взгляд второй вариант яснее, хотя и менее универсален. Почему? об этом чуть ниже.

Типичные ошибки: неявное перекодирование

Неявное перекодирование

```
>>> unicode(regular_string)
UnicodeDecodeError: 'ascii' codec can't decode byte ...
```

- Часто встречается: у англоязычных разработчиков
- *Неправильное решение*: исправить `'ascii'` в `site.py` на используемую кодировку
- **Правильное решение**: использовать явную перекодировку

  ```
  >>> unicode(regular_string, 'utf-8')
  ```

  либо

  ```
  >>> regular_string.decode('utf-8')
  ```

Следующая типичная ошибка - программа ожидает юникод, или ожидает строку, но не

проверяет типы данных. Ошибки - либо UnicodeDecodeError, либо UnicodeEncodeError. Это как раз к тому, почему метод .decode менее универсален.

Заметьте, тут не TypeError, не ValueError, чем и смущают неопытных разработчиков. Причем исключение зачастую возбуждается на более глубоком уровне, чем допущена ошибка типов. В этом случае следует внимательней проанализировать типы передаваемых данных. А на будущее - проверять типы поступаемых данных.

Типичные ошибки: не тот тип данных

Не тот тип данных

```
>>> 'ожидается юникод'.encode('utf-8')
UnicodeDecodeError: 'ascii' codec can't decode byte ...

>>> u'ожидается строка'.decode('utf-8')
UnicodeEncodeError: 'ascii' codec can't encode characters ...
```

- Часто встречается: у ленивых или невнимательных русскоязычных разработчиков
- *Неправильное решение*: исправить `'ascii'` в `site.py` на используемую кодировку
- **Правильное решение**: проверять типы данных

  ```
  >>> isinstance('ожидается юникод', unicode)
  False
  ```

Выше уже говорил о неправильных способах решения проблем с юникод. На первом месте стоит правка site.py. Ниже перечислены основные причины, почему так делать нельзя. В первую очередь, это конечно потеря переносимости и ложное чувство рабочей программы.

Типичные ошибки: изменение `site.py`

Изменение `site.py` — «универсальное» решение проблем

- Часто встречается: у ленивых русскоязычных разработчиков
- *Неправильное решение*: исправить `'ascii'` в `site.py` на используемую кодировку
  - Теряется переносимость программы
    - Зависимость поведения программы от окружения
    - Завязка на конкретную кодировку
  - Ложное чувство правильности работы программы
- **Правильное решение**: использовать явную перекодировку, указывать англоязычным разработчикам на ошибки использования юникод

7 Всё

Спасибо

Спасибо за внимание

Вопросы?