20 Юникод

Строки уже неоднократно встречались вам в этой книге, но мы еще не обсуждали одно из самых больших изменений Python 3 — строки Юникода. В Python 2 строки Юникода поддерживались, но их нужно было специально создавать. Теперь ситуация изменилась, все строки хранятся в Юникоде.

20.1. Историческая справка

Что такое Юникод (Unicode)? Это стандарт представления глифов (символы, входящие в большинство письменных языков, а также знаки и эмодзи). Спецификацию стандарта можно найти на сайте Юникода[1], причем стандарт часто обновляется. Стандарт состоит из различных документов и диаграмм, связывающих *кодовые пункты* (шестнадцатеричные числа, такие как 0048 или 1F600) с глифами (H или ☺) и именами (`LATIN CAPITAL H` и `GRINNING FACE`). Кодовые пункты и имена уникальны, хотя многие глифы кажутся очень похожими.

А теперь небольшая историческая справка. С широким распространением компьютеров разные организации стали предлагать разные схемы для отображения двоичных данных на строковые данные. Одна из таких *кодировок* — ASCII — использует 7 битов данных для отображения на 128 знаков и управляющих символов. Такая кодировка нормально работает в средах, ориентированных на латинский алфавит, — напри-

[1] https://unicode.org

мер, в английском языке. Наличие 128 разных глифов предоставляет достаточно места для символов нижнего регистра, символов верхнего регистра, цифр и знаков препинания.

Со временем поддержка языков, отличных от английского, стала более распространенной, и кодировки ASCII оказалось недостаточно. В семействе Windows до Windows 98 поддерживается кодировка Windows-1252 с поддержкой различных символов с диакритикой и знаков (например, знака евро).

Все эти схемы кодирования обеспечивают однозначное отображение байтов на символы. Для поддержки китайской, корейской и японской письменности потребуется много более 128 символов. Четырехбайтовая кодировка позволяет поддерживать свыше 4 миллиардов символов. Тем не менее такая универсальность не дается даром. Для большинства людей, использующих только символы из кодировки ASCII, четырехкратное увеличение затрат памяти при таком же объеме данных выглядит колоссальными потерями памяти.

Чтобы иметь возможность поддержки всех символов без лишних затрат памяти, пришлось пойти на компромисс — отказаться от кодирования символов последовательностью битов. Вместо этого символы были абстрагированы. Каждый символ отображался на уникальный *кодовый пункт* (обладающий шестнадцатеричным значением и уникальным именем). Затем различные кодировки отображали кодовые пункты на битовые кодировки. Юникод устанавливает соответствие между символом и кодовым пунктом, а не конкретным представлением. В разных контекстах альтернативные представления могут обеспечивать более эффективные характеристики.

Одна из кодировок, UTF-32, использует 4 байта для хранения информации о символе. Такое представление удобно для низкоуровневых программистов благодаря тривиальным операциям индексирования. С другой стороны, она расходует вчетверо больше памяти, чем ASCII, для кодирования серии букв латинского алфавита.

Концепция кодировок переменной ширины также помогала бороться с затратами памяти. Одной из таких кодировок является UTF-8, в которой для представления символа используется от 1 до 4 байтов. Кроме того, UTF-8 обладает обратной совместимостью с ASCII. UTF-8 — самая

Рис. 20.1. Чтение кодовых таблиц на сайте unicode.org. В таблицах перечислены глифы с соответствующими шестнадцатеричными кодовыми пунктами. За этой таблицей следует другая таблица с кодовым пунктом, глифом и названием. Вы можете использовать глиф, имя или кодовый пункт. Если кодовый пункт содержит более 4 цифр, используйте прописную букву *U* и дополните код слева нулями до 8 цифр. Если же цифр 4 и меньше, необходимо использовать строчную букву *u* и дополнять код слева нулями до 4 цифр. Также приведен пример декодирования байтовой строки UTF-8 в соответствующий глиф

распространенная кодировка в интернете — отлично подходит для кодирования символов, так как она поддерживается многими приложениями и операционными системами.

СОВЕТ

Чтобы узнать предпочтительную кодировку для вашей машины, выполните следующий код (приведен результат для моего 2015 Mac):

```
>>> import locale
>>> locale.getpreferredencoding(False)
'UTF-8'
```

Еще раз проясним: UTF-8 — кодировка байтов кодовых пунктов Юникода. Заявить, что UTF-8 и Юникод — одно и то же, в лучшем случае неточность, а в худшем — демонстрация непонимания способа кодирования символов. Более того, само название происходит от слов «Unicode Transformation Format — 8 bit», то есть «формат преобразования Юникода — 8-разрядный», то есть это формат для Юникода.

Рассмотрим пример. Символ с именем `SUPERSCRIPT TWO` определяется в стандарте Юникода как кодовый пункт U+00b2. Его глиф (печатное представление) имеет вид ². В ASCII этот символ представить невозможно. В Windows-1252 такое представление существует, символ кодируется байтом *b2* в шестнадцатеричной записи (это представление совпадает с кодовым пунктом, хотя такое совпадение не гарантируется). В UTF-8 ему соответствует кодировка *c2*. Аналогичным образом в UTF-16 используется кодировка *ff fe b2 00*. Таким образом, у этого кодового пункта Юникода существует несколько разных кодировок.

20.2. Основные этапы в Python

В Python можно создавать строки Юникода. Собственно, вы делали это с первых страниц книги: напомним, что в Python 3 все строки кодируются в Юникоде. Но что делать, если вы хотите создать строку с символами, не входящими в ASCII? Ниже приведен код создания строки «x^2». Если вы найдете глиф, который должен использоваться в строке, скопируйте символ в код:

```
>>> result = 'x²'
```

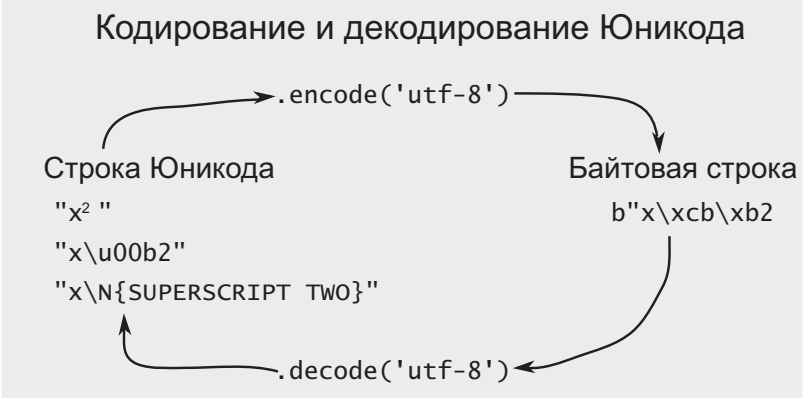

Рис. 20.2. *Кодирование* (в данном случае с использованием UTF-8) строки Юникода в байтовое представление и последующее *декодирование* той же байтовой строки в Юникод (тоже с использованием UTF-8). Также при декодировании следует предельно четко выражать свои намерения, потому что неточность приведет к появлению ошибочных данных или искажению символов

Такой способ обычно работает, хотя могут возникнуть проблемы, если ваш шрифт не поддерживает указанный глиф. Тогда вместо него будет отображаться так называемый «*тофу*» (пустой прямоугольник или ромб с вопросительным знаком). Также для включения символов, не входящих в ASCII, можно включить шестнадцатеричный кодовый пункт Юникода после префикса `\u`:

```
>>> result2 = 'x\u00b2'
```

Обратите внимание: эта строка тождественна предыдущей строке:

```
>>> result == result2
True
```

Наконец, можно использовать в строке имя кодового пункта, заключив его в фигурные скобки в конструкции `\N{}`:

```
>>> result3 = 'x\N{SUPERSCRIPT TWO}'
```

Все эти способы работают. И все они возвращают одну и ту же строку Юникода:

```
>>> print(result, result2, result3)
x² x² x²
```

Третий вариант получается менее компактным. Но если вы не помните нужный кодовый пункт или глиф не поддерживается шрифтом, пожалуй, этот вариант оказывается самым удобочитаемым.

СОВЕТ

В документации Python имеется раздел, посвященный Юникоду. Введите `help()`, а затем `UNICODE`. В этом разделе обсуждается не столько Юникод, сколько различные способы создания строк Python.

20.3. Кодирование

Пожалуй, один из ключей к пониманию Юникода в Python — понимание того, что *строка Юникода кодируется в байтовую строку*. Байтовые строки никогда не кодируются, но могут декодироваться в строку Юникода. Аналогичным образом *строки Юникода не декодируются*. Также на процессы кодирования и декодирования можно взглянуть под другим углом: кодирование преобразует понятное или осмысленное для человека представление в абстрактное представление, предназначенное для хранения (Юникод в байты или буквы в байты), а декодирование преобразует это абстрактное представление обратно в форму, удобную для человека.

Для заданной строки Юникода можно вызвать метод `.encode`, чтобы просмотреть ее представление в различных кодировках. Начнем с UTF-8:

```
>>> x_sq = 'x\u00b2'
>>> x_sq.encode('utf-8')
b'x\xc2\xb2'
```

Если Python не поддерживает кодировку, будет выдана ошибка `UnicodeEncodeError`. Это означает, что кодовые пункты Юникода не поддерживаются в этой кодировке. Например, ASCII не поддерживает символ возведения в квадрат:

```
>>> x_sq.encode('ascii')
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'ascii' codec can't encode character
'\xb2' in position 1: ordinal not in range(128)
```

Если вы достаточно давно работаете с Python, вероятно, вы сталкивались с этой ошибкой. Она означает, что указанное кодирование не позволит представить все символы. Если вы уверены в том, что хотите заставить Python провести кодирование, то можете передать параметр `errors`. Чтобы проигнорировать символы, которые Python не может представить, передайте параметр `errors='ignore'`:

```
>>> x_sq.encode('ascii', errors='ignore')
b'x'
```

Если передать параметр `errors='replace'`, Python вставит вопросительные знаки вместо неподдерживаемых байтов:

```
>>> x_sq.encode('ascii', errors='replace')
b'x?'
```

ПРИМЕЧАНИЕ

Модуль `encodings` содержит отображения для кодировок, поддерживаемых Python. В Python 3.6 поддерживаются 99 кодировок. Многие современные приложения стараются ограничиваться UTF-8, но если вам нужна поддержка других кодировок, вполне возможно, что они поддерживаются в Python.

Таблица кодировок находится в encodings.aliases.aliases. Кроме того, таблица кодировок приведена в документации модуля на сайте Python[1].

Несколько возможных вариантов кодирования для этой строки:

```
>>> x_sq.encode('cp1026') # Турецкий
b'\xa7\xea'
>>> x_sq.encode('cp949') # Корейский
b'x\xa9\xf7'
>>> x_sq.encode('shift_jis_2004') # Японский
b'x\x85K'
```

Хотя Python поддерживает много кодировок, они все реже используются на практике. Как правило, они встречаются только в унаследованных приложениях. В наши дни большинство приложений использует UTF-8.

[1] https://docs.python.org/3/library/codecs.html