
유니코드 HOWTO

출시 버전 **3.7.16**

**Guido van Rossum
and the Python development team**

2월 08, 2023

Contents

1	유니코드 소개	2
1.1	정의	2
1.2	인코딩	3
1.3	참조	4
2	파이썬의 유니코드 지원	4
2.1	문자열 형	4
2.2	바이트열로 변환	5
2.3	파이썬 소스 코드에서 유니코드 리터럴	6
2.4	유니코드 속성	7
2.5	Comparing Strings	7
2.6	유니코드 정규식	8
2.7	참조	9
3	유니코드 데이터 읽고 쓰기	9
3.1	유니코드 파일 이름	10
3.2	유니코드 인식 프로그램 작성 팁	11
3.3	참조	12
4	감사 인사	12
	색인	13

버전 1.12

This HOWTO discusses Python's support for the Unicode specification for representing textual data, and explains various problems that people commonly encounter when trying to work with Unicode.

1 유니코드 소개

1.1 정의

Today's programs need to be able to handle a wide variety of characters. Applications are often internationalized to display messages and output in a variety of user-selectable languages; the same program might need to output an error message in English, French, Japanese, Hebrew, or Russian. Web content can be written in any of these languages and can also include a variety of emoji symbols. Python's string type uses the Unicode Standard for representing characters, which lets Python programs work with all these different possible characters.

Unicode (<https://www.unicode.org/>) is a specification that aims to list every character used by human languages and give each character its own unique code. The Unicode specifications are continually revised and updated to add new languages and symbols.

A **character** is the smallest possible component of a text. 'A', 'B', 'C', etc., are all different characters. So are 'È' and 'Í'. Characters vary depending on the language or context you're talking about. For example, there's a character for "Roman Numeral One", 'I', that's separate from the uppercase letter 'I'. They'll usually look the same, but these are two different characters that have different meanings.

The Unicode standard describes how characters are represented by **code points**. A code point value is an integer in the range 0 to 0x10FFFF (about 1.1 million values, with some 110 thousand assigned so far). In the standard and in this document, a code point is written using the notation U+265E to mean the character with value 0x265e (9,822 in decimal).

The Unicode standard contains a lot of tables listing characters and their corresponding code points:

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

엄밀히 말하자면, 이러한 정의는 '이것이 문자 U+265E'라고 말하는 것은 의미가 없음을 뜻합니다. U+265E는 어떤 특정 문자를 표현하는 코드 포인트일 뿐입니다; 이 경우에는 'BLACK CHESS KNIGHT'('♞')를 나타냅니다. 일상적인 문맥에서 코드 포인트와 문자 사이의 구분은 때때로 잊힐 겁니다.

문자는 화면이나 종이에 **글리프**라 불리는 그래픽 요소의 집합으로 표시됩니다. 예를 들어 대문자 A의 글리프는 두 개의 대각 획과 한 개의 수평 획이지만, 정확한 세부사항은 글꼴에 따라 다릅니다. 대부분의 파이썬 코드는 글리프를 걱정할 필요가 없습니다; 표시할 올바른 글리프를 찾는 것은 일반적으로 GUI 툴킷이나 터미널의 글꼴 렌더러의 일입니다.

1.2 인코딩

To summarize the previous section: a Unicode string is a sequence of code points, which are numbers from 0 through $0 \times 10FFFF$ (1,114,111 decimal). This sequence of code points needs to be represented in memory as a set of **code units**, and **code units** are then mapped to 8-bit bytes. The rules for translating a Unicode string into a sequence of bytes are called a **character encoding**, or just an **encoding**.

The first encoding you might think of is using 32-bit integers as the code unit, and then using the CPU's representation of 32-bit integers. In this representation, the string "Python" might look like this:

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

이 표현은 간단하지만 여러 문제를 가지고 있습니다.

1. 이식성이 없습니다; 다른 프로세서는 바이트를 다르게 정렬합니다.
2. 이는 공간을 매우 낭비하는 겁니다. 대부분 텍스트에서 주요 코드 포인트는 127 또는 255보다 작으므로 많은 공간이 0×00 으로 채워집니다. 위의 문자열은 ASCII 표현에 필요한 6 바이트와 비교하여 24 바이트를 사용합니다. 늘어난 RAM 사용은 중요하지 않지만(데스크톱 컴퓨터는 기가바이트 단위의 RAM을 갖고 있고, 문자열은 대개 그 정도로 크지 않습니다), 디스크와 네트워크 대역폭 사용을 4배로 확장하는 것은 용납할 수 없습니다.
3. `strlen()`과 같은 기존의 C 함수와 호환이 안 되기 때문에 새로운 와이드 문자열 함수 계열을 사용해야 합니다.

Therefore this encoding isn't used very much, and people instead choose other encodings that are more efficient and convenient, such as UTF-8.

UTF-8 is one of the most commonly used encodings, and Python often defaults to using it. UTF stands for "Unicode Transformation Format", and the '8' means that 8-bit values are used in the encoding. (There are also UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) UTF-8 uses the following rules:

1. 만약 코드 포인트가 128보다 작다면, 해당 바이트 값으로 표현됩니다.
2. 만약 코드 포인트가 128보다 크거나 같다면, 시퀀스의 각 바이트가 128에서 255 사이인 둘, 셋 또는 네 개의 바이트 시퀀스로 바뀝니다.

UTF-8은 몇 가지 편리한 특징이 있습니다:

1. 모든 유니코드 코드 포인트를 처리 할 수 있습니다.
2. A Unicode string is turned into a sequence of bytes that contains embedded zero bytes only where they represent the null character (U+0000). This means that UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can't handle zero bytes for anything other than end-of-string markers.
3. ASCII 텍스트의 문자열 역시 유효한 UTF-8 텍스트입니다.
4. UTF-8은 꽤 알차다; 일반적으로 사용되는 문자 대부분을 한두 개의 바이트로 표현할 수 있습니다.
5. 만약 바이트가 손상 또는 손실되었다면, 다음 UTF-8 인코딩 코드 포인트의 시작을 결정하고 다시 동기화할 수 있습니다. 무작위 8비트 데이터가 유효한 UTF-8처럼 보일 가능성은 낮습니다.
6. UTF-8 is a byte oriented encoding. The encoding specifies that each character is represented by a specific sequence of one or more bytes. This avoids the byte-ordering issues that can occur with integer and word oriented encodings, like UTF-16 and UTF-32, where the sequence of bytes varies depending on the hardware on which the string was encoded.

1.3 참조

유니코드 컨소시엄 사이트는 유니코드 사양의 문자 차트, 용어집 그리고 PDF 버전의 유니코드 명세를 갖고 있습니다. 어려운 읽기를 준비하세요. 유니코드의 기원과 개발의 연대기 역시 이 사이트에서 볼 수 있습니다.

On the Computerphile Youtube channel, Tom Scott briefly *discusses the history of Unicode and UTF-8* <<https://www.youtube.com/watch?v=MijmeoH9LT4>> (9 minutes 36 seconds).

Jukka Korpela는 표준을 이해할 수 있도록 유니코드 문자표 읽기에 대한 입문 안내서를 작성했습니다.

또 다른 좋은 입문 글을 Joel Spolsky가 썼습니다. 이 입문서로도 명확하지 않은 경우 계속하기 전에 이 대체 문서를 읽으세요.

위키피디아 항목은 때때로 도움이 됩니다; 예를 들어, “문자 인코딩”과 UTF-8 항목을 보세요.

2 파이썬의 유니코드 지원

이제 유니코드의 기초를 배웠으므로 파이썬의 유니코드 기능을 살펴볼 수 있습니다.

2.1 문자열 형

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

파이썬 소스 코드의 기본 인코딩은 UTF-8이므로 문자열 리터럴에 유니코드 문자를 쉽게 포함할 수 있습니다:

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

사이드 노트: 파이썬 3은 유니코드 문자를 식별자에서도 지원합니다:

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

편집기에서 특정 문자를 입력 할수 없거나 어떤 이유에서 ASCII만으로 소스 코드를 작성하고자 한다면, 문자열 리터럴에 이스케이프 시퀀스를 사용할 수 있습니다. (시스템에 따라 다르지만, `u` 이스케이프 대신에 진짜 대문자 델타 클리프가 나타날 수 있습니다.)

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

추가로, 문자열을 `bytes`의 `decode()` 메서드를 사용하여 만들 수 있습니다. 이 메서드는 UTF-8과 같은 *encoding* 인자와 선택적으로 *errors* 인자를 받습니다.

errors 인자는 인코딩의 규칙에 따라 입력 문자열을 변환할 수 없는 경우의 응답을 지정합니다. 이 인자의 유효한 값은 `'strict'` (UnicodeDecodeError 예외 발생), `'replace'` (U+FFFD, REPLACEMENT CHARACTER

사용), 'ignore' (유니코드 결과에서 문자를 그냥 생략), 또는 'backslashreplace' (\xNN 이스케이프 시퀀스를 삽입)입니다. 다음 예제는 그 차이점을 보여줍니다:

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

인코딩은 인코딩의 이름을 포함하는 문자열로 지정됩니다. 파이썬에는 대략 100개의 서로 다른 인코딩이 있습니다. 리스트는 `standard-encodings`에서 파이썬 라이브러리 레퍼런스를 보세요. 어떤 인코딩은 다양한 이름을 갖습니다; 예를 들어, 'latin-1', 'iso_8859_1' 그리고 '8859'는 전부 같은 인코딩의 동의어입니다.

한 문자 유니코드 문자열은 정수를 받고 해당 코드 포인트를 포함하는 길이 1인 유니코드 문자열을 반환하는 `chr()` 내장 함수로도 만들 수 있습니다. 반대 작업은 한 문자 유니코드 문자열을 받고 코드 포인트 값을 반환하는 `ord()` 내장 함수입니다:

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

2.2 바이트열로 변환

`bytes.decode()`의 반대 메서드는 요청된 *encoding*으로 인코딩 된 유니코드 문자열의 `bytes`를 반환하는 `str.encode()`입니다.

`errors` 매개변수는 `decode()` 메서드의 매개변수와 같지만 가능한 핸들러를 조금 더 제공합니다. 'strict', 'ignore', 'replace' (이 경우 인코딩할 수 없는 문자 대신에 물음표를 삽입)뿐만 아니라 'xmlcharrefreplace' (XML 문자 참조 삽입), `backslashreplace` (\uNNNN 이스케이프 시퀀스 삽입) 그리고 `namereplace` (\N{...} 이스케이프 시퀀스 삽입)도 있습니다.

아래 예제에서 서로 다른 결과를 보여줍니다:

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

(다음 페이지에 계속)

```
>>> u.encode('ascii', 'namereplace')
b'\\N{YI SYLLABLE IT}abcd\\u07b4'
```

사용 가능한 인코딩을 등록하고 접근할 수 있도록 하는 저수준 루틴은 `codecs` 모듈에서 찾을 수 있습니다. 새로운 인코딩을 구현하는 것도 `codecs` 모듈의 이해가 필요합니다. 하지만 이 모듈에 의해서 반환되는 인코딩과 디코딩 함수는 대체로 편만한 것보다는 저수준이며, 새로운 인코딩을 작성하는 것은 전문적인 작업이므로 HOWTO에서 이 모듈을 다루지는 않겠습니다.

2.3 파이썬 소스 코드에서 유니코드 리터럴

파이썬 소스 코드에서 특정한 유니코드 코드 포인트는 `\u` 이스케이프 시퀀스로 쓸 수 있으며, 코드 포인트를 의미하는 네 개의 16진수가 뒤따릅니다. `\U` 이스케이프 시퀀스와 비슷하지만 네 개가 아닌 여덟 개의 숫자여야 합니다:

```
>>> s = "a\xac\u1234\u20ac\u00008000"
... #      ^^^^ two-digit hex escape
... #      ^^^^^ four-digit Unicode escape
... #      ^^^^^^^^^ eight-digit Unicode escape
>>> [ord(c) for c in s]
[97, 172, 4660, 8364, 32768]
```

127보다 큰 코드 포인트에 대해 이스케이프 시퀀스를 사용하는 것은 양이 적을 때 괜찮지만, 프랑스어나 다른 악센트를 사용하는 언어로 작성된 프로그램 메시지 같이 악센트가 있는 문자를 사용할 경우 성가십니다. `chr()` 내장 함수를 이용하여 문자열을 조립할 수는 있지만 더 짜증납니다.

사용하는 언어의 자연스러운 인코딩으로 리터럴을 쓸 수 있어야 이상적입니다. 그래야 악센트가 있는 문자를 자연스럽게 표시하는 사용자가 가장 좋아하는 편집기로 파이썬 소스 코드를 편집할 수 있고, 실행시점에 올바른 문자를 가질 수 있습니다.

파이썬은 소스 코드를 UTF-8로 작성하는 것을 기본으로 지원하지 않지만, 원하는 인코딩을 선언한다면 거의 모든 인코딩을 쓸 수 있습니다. 이는 소스 파일의 첫 번째 또는 두 번째 줄에 특별한 주석을 포함해 작동합니다:

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

u = 'abcdé'
print(ord(u[-1]))
```

이 문법은 파일에 지역적인 변수를 명시하는데 사용되는 이맥스 표기법에서 영감을 받았습니다. 이맥스는 수많은 서로 다른 변수를 지원하지만 파이썬은 오직 ‘coding’만을 지원합니다. `-*-` 기호는 이맥스에게 주석이 특별함을 나타냅니다; 파이썬에게 아무 의미도 없지만 그저 관례를 따르는 것뿐입니다. 파이썬은 주석에서 `coding: name`이나 `coding=name`을 찾습니다.

이러한 주석을 포함하지 않는다면, 이미 언급한 것처럼 기본 인코딩으로 UTF-8이 사용됩니다. **PEP 263**에서 정보를 더 보시기 바랍니다.

2.4 유니코드 속성

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

아래의 프로그램은 몇몇 개의 문자에 대한 정보를 표시하고 특정한 문자의 숫자 값을 출력합니다:

```
import unicodedata

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

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

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

실행했을 때 다음을 출력합니다:

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

카테고리 코드는 문자의 유형을 설명하는 약어입니다. 이것들은 “Letter”, “Number”, “Punctuation”, 또는 “Symbol”과 같은 카테고리로 묶여 있고, 이는 하위 카테고리로 나누어 집니다. 위 결과물에서 코드를 가져와 보면, 'Ll'는 ‘Letter, lowercase’, 'No'는 “Number, other”, 'Mn'은 “Mark, nonspacing”, 그리고 'So'는 “Symbol, other”를 뜻합니다. 카테고리 코드 목록에 대해서는 유니코드 문자 데이터베이스 문서의 일반 카테고리값 섹션을 보세요.

2.5 Comparing Strings

Unicode adds some complication to comparing strings, because the same set of characters can be represented by different sequences of code points. For example, a letter like ‘ê’ can be represented as a single code point U+00EA, or as U+0065 U+0302, which is the code point for ‘e’ followed by a code point for ‘COMBINING CIRCUMFLEX ACCENT’. These will produce the same output when printed, but one is a string of length 1 and the other is of length 2.

One tool for a case-insensitive comparison is the `casefold()` string method that converts a string to a case-insensitive form following an algorithm described by the Unicode Standard. This algorithm has special handling for characters such as the German letter ‘ß’ (code point U+00DF), which becomes the pair of lowercase letters ‘ss’.

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

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

```
import unicodedata
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

    return NFD(s1) == NFD(s2)

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

실행했을 때 다음을 출력합니다:

```
$ python3 compare-strs.py
length of first string= 1
length of second string= 2
True
```

The first argument to the `normalize()` function is a string giving the desired normalization form, which can be one of 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode Standard also specifies how to do caseless comparisons:

```
import unicodedata

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

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

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

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

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

2.6 유니코드 정규식

`re` 모듈이 지원하는 정규식은 바이트열 또는 문자열로 제공됩니다. `\d`와 `\w` 같은 특별한 문자 시퀀스 몇몇은 패턴이 바이트열 또는 문자열에 의해 지원되는지 여부에 따라 다른 의미가 있습니다. 예를 들어, 바이트열에서 `\d`는 `[0-9]`와 일치하지만, 문자열에서는 'Nd' 카테고리에 속하는 아무 문자와 일치합니다.

이 예제의 문자열은 태국과 아라비아 숫자로 쓰인 숫자 57을 갖고 있습니다:

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

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


실행했을 때, `\d+`은 태국 숫자와 일치하고 출력합니다. `re.ASCII` 플래그를 `compile()`에 제공했을 경우, `\d+`은 부분 문자열 “57”을 대신 일치시킵니다.

비슷하게, `\w`는 매우 다양한 유니코드와 일치하지만, 바이트열이거나 `re.ASCII`가 제공되면 오직 `[a-zA-Z0-9_]`과 일치하고, `\s`는 유니코드 공백 문자나 `[\t\n\r\f\v]`와 일치합니다.

2.7 참조

파이썬 유니코드 지원에 대한 몇 개의 좋은 대안 토론은 다음과 같습니다:

- Nick Coghlan의 [파이썬 3의 텍스트 파일 처리](#).
- Ned Batchelder가 PyCon 2012에서 발표한 [실용 유니코드](#).

`str` 타입은 파이썬 라이브러리 레퍼런스 `textseq`에 설명되어 있습니다.

`unicodedata` 모듈에 대한 문서입니다.

`codecs` 모듈에 대한 문서입니다.

Marc-André Lemburg는 EuroPython 2002에서 발표한 “파이썬과 유니코드”라는 제목의 [프레젠테이션 \(PDF 슬라이드\)](#)을 주었습니다. 이 슬라이드는 파이썬 2의 유니코드 기능(유니코드 문자열 타입을 `unicode`라 부르코 리터럴 `u`로 시작하는) 디자인에 대한 훌륭한 개요서입니다.

3 유니코드 데이터 읽고 쓰기

유니코드 데이터로 동작하는 코드를 작성했다면, 다음은 입출력이 문제입니다. 프로그램에 유니코드 문자열을 어떻게 집어넣을 것인가, 그리고 유니코드를 어떻게 저장 또는 전송에 적합한 형식으로 유니코드를 전환할 것인가?

입력 소스 및 출력 대상에 따라 아무것도 할 필요 없을 수도 있습니다; 응용 프로그램에서 사용되는 라이브러리가 유니코드를 기본 지원하는지 확인해야 됩니다. 예를 들어, XML 파서는 종종 유니코드 데이터를 반환합니다. 많은 관계형 데이터베이스 역시 유니코드 값 칼럼을 지원하고, SQL 질의로부터 유니코드 값을 반환받을 수 있습니다.

유니코드 데이터는 디스크에 쓰이거나 소켓에 전달되기 전에 보통 특정 인코딩으로 변경됩니다. 다음의 모든 작업을 직접 할 수 있습니다: 파일 열기, 8-bit 바이트열 객체 읽기, 그리고 `bytes.decode(encoding)`로 바이트열 전환하기. 하지만 이러한 수동 접근은 권장하지 않습니다.

한 가지 문제는 인코딩의 멀티 바이트 특성입니다; 하나의 유니코드 문자는 여러 바이트로 표현됩니다. 임의의 크기의 청크로 파일을 읽으려면 (1024 또는 4096바이트라 할 때), 단일 유니코드 문자를 인코딩하는 바이트 일부만 청크 끝에 읽는 경우를 잡기 위해 예러 처리 코드를 작성해야 합니다. 한 가지 해결책은 파일 전체를 메모리에 읽고 디코딩을 수행하는 것이지만, 극도로 큰 파일로 작업하는 것을 방해합니다; 2기가바이트 파일을 읽어야만 한다면, 2기가바이트 메모리가 필요합니다. (인코딩된 문자열과 유니코드 버전을 잠깐 메모리에 가지고 있어야 하므로 실제로는 더 필요합니다.)

부분 코딩 시퀀스의 경우를 잡기 위해 저수준 디코딩 인터페이스를 사용하는 것이 해결방법입니다. 이를 구현하는 작업은 이미 수행되었습니다: 내장 `open()` 함수는 파일 내용물이 지정된 인코딩이라고 가정하고, `read()` 또는 `write()`와 같은 메서드에서 유니코드 매개변수를 받아들이는 파일 객체를 반환할 수 있습니다. 이는 `str.encode()`와 `bytes.decode()`에서처럼 해석되는 `open()`의 `encoding`과 `errors` 매개변수를 통해 작동합니다.

그러므로 파일에서 유니코드를 읽는 것은 간단합니다:

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

업데이트 모드로 파일을 열어 읽기 또는 쓰기를 할 수도 있습니다:

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

유니코드 문자 U+FEFF는 바이트 순서 표시(BOM)로 사용되고, 파일의 바이트 순서를 자동감지하기 위해 파일의 맨 처음 문자로 쓰이기도 합니다. UTF-16과 같은 일부 인코딩에서는 파일 시작 부분에 BOM이 있어야 합니다; 이러한 인코딩이 쓰일 때, BOM이 자동으로 첫번째 문자로 작성되고 파일을 읽을 때 조용히 없어집니다. 리틀 엔디안과 빅 엔디안을 위해 특정 바이트 순서를 지정하고 BOM을 생략하지 않는 'utf-16-le'와 'utf-16-be' 같은 인코딩의 변종들이 있습니다.

일부 영역에서는 UTF-8로 인코딩된 파일의 시작 부분에 “BOM”을 사용하는 것이 관례이기도 합니다; UTF-8은 바이트 순서에 의존하지 않으므로 이 이름은 오해의 소지가 있습니다. 이 표시는 이 파일이 UTF-8로 인코딩했음을 단순히 알립니다. 그런 파일을 읽으려면 자동으로 이 표시를 건너뛰기 위해 'utf-8-sig' 코덱을 사용하세요.

3.1 유니코드 파일 이름

Most of the operating systems in common use today support filenames that contain arbitrary Unicode characters. Usually this is implemented by converting the Unicode string into some encoding that varies depending on the system. Today Python is converging on using UTF-8: Python on MacOS has used UTF-8 for several versions, and Python 3.6 switched to using UTF-8 on Windows as well. On Unix systems, there will only be a filesystem encoding if you've set the LANG or LC_CTYPE environment variables; if you haven't, the default encoding is again UTF-8.

인코딩을 수동으로 하고 싶을 때를 대비하여 `sys.getfilesystemencoding()` 함수는 현재 사용하고 있는 시스템의 인코딩을 반환하지만, 귀찮게 그럴 이유는 없습니다. 읽기 또는 쓰기를 위해 파일을 열 때, 보통 파일 이름으로 유니코드 문자열을 제공하기만 하면 되고, 자동으로 올바른 인코딩으로 변환됩니다:

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

os 모듈 안의 `os.stat()` 과 같은 함수 역시 유니코드 파일 이름을 수용합니다.

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

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

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

아래의 출력을 만들 것입니다:

```
$ python listdir-test.py
[b'filename\xe4\x94\x80abc', ...]
['filename\u4500abc', ...]
```

첫번째 리스트는 UTF-8로 인코딩된 파일 이름을 갖고, 두 번째 리스트는 유니코드 버전을 갖습니다.

Note that on most occasions, you should can just stick with using Unicode with these APIs. The bytes APIs should only be used on systems where undecodable file names can be present; that's pretty much only Unix systems now.

3.2 유니코드 인식 프로그램 작성 팁

이 섹션은 유니코드를 다루는 소프트웨어를 작성할 때의 몇 가지 제안을 제공합니다.

가장 중요한 팁은:

소프트웨어는 가능한 한 빨리 입력 데이터를 디코딩하고 마지막에만 출력을 인코딩하여 내부적으로 유니코드 문자열에서만 작동하는 것이 좋습니다.

유니코드와 바이트 문자열을 수용하는 처리 함수를 작성하려고 시도한다면, 두 가지 다른 종류의 문자열을 결합할 때마다 프로그램이 버그에 취약하다는 것을 알 수 있습니다. 자동 인코딩이나 디코딩은 없습니다: `str + bytes`을 수행한다면 `TypeError`가 발생합니다.

웹 브라우저나 다른 신뢰할 수 없는 소스로부터 온 데이터를 사용할 때, 일반적인 기법은 생성된 명령행에서 문자열을 사용하거나 데이터베이스에 저장하기 전에 문자열에서 잘못된 문자를 검사하는 것입니다. 이렇게 하고 있다면, 인코딩된 바이트열 데이터가 아닌 디코딩 된 문자열을 검사하도록 조심하기 바랍니다; 어떤 인코딩은 일대일 대응되지 않거나 ASCII와 완전히 호환되지 않는 흥미로운 속성을 가지고 있습니다. 입력 데이터가 인코딩을 지정하는 경우 특히 그러한데, 공격자가 인코딩한 바이트 스트림 안에 악의적인 텍스트를 숨기는 방법을 택할 수 있기 때문입니다.

파일 인코딩끼리 변환

`StreamRecoder` 클래스는 인코딩 #1으로 데이터를 반환하는 스트림을 받아서 인코딩 #2로 데이터를 반환하는 스트림처럼 동작하여 인코딩 간에 투명하게 변환할 수 있습니다.

예를 들어, Latin-1을 사용하는 `f` 입력 파일을 가지고 있다면, 이를 UTF-8로 인코딩한 바이트열을 반환하기 위해 `StreamRecoder`로 감쌀 수 있습니다:

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

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

알 수 없는 인코딩의 파일

파일을 변경해야 하지만 파일의 인코딩을 모를 때 할 수 있는 일은 무엇일까요? 인코딩이 ASCII 호환이라는 것을 알고 있고 ASCII 일부분을 검토 또는 수정만 하고자 한다면, `surrogateescape` 에러 핸들러와 함께 파일을 열 수 있습니다:

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

# make changes to the string 'data'

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

The `surrogateescape` error handler will decode any non-ASCII bytes as code points in a special range running from U+DC80 to U+DCFF. These code points will then turn back into the same bytes when the `surrogateescape` error handler is used to encode the data and write it back out.

3.3 참조

David Beazley가 PyCon 2010에서 발표한 파이썬3 입출력 마스터하기의 한 섹션은 텍스트 처리와 바이너리 데이터 처리에 대해 논의합니다.

Marc-André Lemburg의 프레젠테이션 PDF 슬라이드 “파이썬에서 유니코드 인식 프로그램 작성”은 문자 인코딩에 관한 질문은 물론 응용 프로그램을 국제화하고 지역화하는 방법에 관해 설명합니다. 이 슬라이드는 파이썬 2.x만을 다룹니다.

PyCon 2013에서 Benjamin Peterson이 발표한 파이썬에서 유니코드의 내부는 파이썬 3.3에서 내부 유니코드 표현에 대해 논의합니다.

4 감사 인사

이 문서의 초안은 Andrew Kuchling이 썼습니다. 이후 Alexander Belopolsky, Georg Brandl, Andrew Kuchling, 그리고 Ezio Melotti에 의해 개정되었습니다.

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

색인

Y

파이썬 향상 제안

PEP 263, 6