

# Text versus bytes

Humans use text. Computers speak bytes<sup>1</sup>.

— Esther Nam and Travis Fischer  
*Character encoding and Unicode in Python*

Python 3 introduced a sharp distinction between strings of human text and sequences of raw bytes. Implicit conversion of byte sequences to Unicode text is a thing of the past. This chapter deals with Unicode strings, binary sequences and the encodings used to convert between them.

Depending on your Python programming context, a deeper understanding of Unicode may or may not be of vital importance to you. In the end, most of the issues covered in this chapter do not affect programmers who deal only with ASCII text. But even if that is your case, there is no escaping the `str` versus `byte` divide. As a bonus, you'll find that the specialized binary sequence types provide features that the “all-purpose” Python 2 `str` type does not have.

In this chapter we will visit the following topics:

- characters, code points and byte representations;
- unique features of binary sequences: `bytes`, `bytearray` and `memoryview`;
- codecs for full Unicode and legacy character sets;
- avoiding and dealing with encoding errors;
- best practices when handling text files;
- the default encoding trap and standard I/O issues;
- safe Unicode text comparisons with normalization;

1. Slide 12 of PyCon 2014 talk *Character encoding and Unicode in Python* ([slides](#), [video](#)).

- utility functions for normalization, case folding and brute-force diacritic removal;
- proper sorting of Unicode text with `locale` and the PyUCA library;
- character metadata in the Unicode database;
- dual-mode APIs that handle `str` and `bytes`;

Let's start with the characters, code points and bytes.

## Character issues

The concept of “string” is simple enough: a string is a sequence of characters. The problem lies in the definition of “character”.

In 2014 the best definition of “character” we have is a Unicode character. Accordingly, the items you get out of a Python 3 `str` are Unicode characters, just like the items of a `unicode` object in Python 2 — and not the raw bytes you get from a Python 2 `str`.

The Unicode standard explicitly separates the identity of characters from specific byte representations.

- The identity of a character — its *code point* — is a number from 0 to 1,114,111 (base 10), shown in the Unicode standard as 4 to 6 hexadecimal digits with a “U+” prefix. For example, the code point for the letter A is U+0041, the Euro sign is U+20AC and the musical symbol G clef is assigned to code point U+1D11E. About 10% of the valid code points have characters assigned to them in Unicode 6.3, the standard used in Python 3.4.
- The actual bytes that represent a character depend on the *encoding* in use. An encoding is an algorithm that converts code points to byte sequences and vice-versa. The code point for A (U+0041) is encoded as the single byte `\x41` in the UTF-8 encoding, or as the bytes `\x41\x00` in UTF-16LE encoding. As another example, the Euro sign (U+20AC) becomes three bytes in UTF-8 — `\xe2\x82\xac` — but in UTF-16LE it is encoded as two bytes: `\xac\x20`.

Converting from code points to bytes is *encoding*; from bytes to code points is *decoding*. See [Example 4-1](#).

*Example 4-1. Encoding and decoding.*

```
>>> s = 'café'
>>> len(s) # ①
4
>>> b = s.encode('utf8') # ②
>>> b
b'caf\xc3\xa9' # ③
>>> len(b) # ④
5
```

```
>>> b.decode('utf8') # ❸  
'café'
```

- ❶ The str 'café' has 4 Unicode characters.
- ❷ Encode str to bytes using UTF-8 encoding.
- ❸ bytes literals start with a b prefix.
- ❹ bytes b has five bytes (the code point for “é” is encoded as two bytes in UTF-8).
- ❺ Decode bytes to str using UTF-8 encoding.



If you need a memory aid to distinguish `.decode()` from `.encode()`, convince yourself that byte sequences can be cryptic machine core dumps while Unicode str objects are “human” text. Therefore, it makes sense that we **decode** bytes to str to get human readable text, and we **encode** str to bytes for storage or transmission.

Although the Python 3 str is pretty much the Python 2 unicode type with a new name, the Python 3 bytes is not simply the old str renamed, and there is also the closely related bytearray type. So it is worthwhile to take a look at the binary sequence types before advancing to encoding/decoding issues.

## Byte essentials

The new binary sequence types are unlike the Python 2 str in many regards. The first thing to know is that there are two basic built-in types for binary sequences: the immutable bytes type introduced in Python 3 and the mutable bytearray, added in Python 2.6<sup>2</sup>.

Each item in bytes or bytearray is an integer from 0 to 255, and not a 1-character string like in the Python 2 str. However a slice of a binary sequence always produces a binary sequence of the same type — including slices of length 1. See [Example 4-2](#).

*Example 4-2. A five-byte sequence as bytes and as bytearray.*

```
>>> cafe = bytes('café', encoding='utf-8') ❶  
>>> cafe  
b'caf\xc3\xa9'  
>>> cafe[0] ❷  
99  
>>> cafe[:1] ❸
```

2. Python 2.6 also introduced bytes, but it's just an alias to the str type, and does not behave like the Python 3 bytes type.

```

b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr ❹
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:] ❺
bytearray(b'\xa9')

```

- ❶ bytes can be built from a `str`, given an encoding.
- ❷ Each item is an integer in `range(256)`.
- ❸ Slices of bytes are also bytes — even slices of a single byte.
- ❹ There is no literal syntax for `bytearray`: they are shown as `bytearray()` with a bytes literal as argument.
- ❺ A slice of `bytearray` is also a `bytearray`.



The fact that `my_bytes[0]` retrieves an `int` but `my_bytes[:1]` returns a bytes object of length 1 should not be surprising. The only sequence type where `s[0] == s[:1]` is the `str` type. Although practical, this behavior of `str` is exceptional. For every other sequence, `s[i]` returns one item, and `s[i:i+1]` returns a sequence of the same type with the `s[i]` item inside it.

Although binary sequences are really sequences of integers, their literal notation reflects the fact that ASCII text is often embedded in them. Therefore, three different displays are used, depending on each byte value:

- For bytes in the printable ASCII range — from space to `~` — the ASCII character itself is used.
- For bytes corresponding to tab, newline, carriage return and `\`, the escape sequences `\t`, `\n`, `\r` and `\\` are used.
- For every other byte value, an hexadecimal escape sequence is used, e.g. `\x00` is the null byte.

That is why in [Example 4-2](#) you see `b'caf\xc3\xa9'`: the first three bytes `b'caf'` are in the printable ASCII range, the last two are not.

Both bytes and `bytearray` support every `str` method except those that do formatting (`format`, `format_map`) and a few others that depend on Unicode data: `casefold`, `isdecimal`, `isidentifier`, `isnumeric`, `isprintable` and `encode`. This means that you can use familiar string methods like `endswith`, `replace`, `strip`, `translate`, `upper` and dozens of others with binary sequences — only using bytes and not `str` arguments. In addition, the regular expression functions in the `re` module also work on binary se-

quences, if the regex is compiled from a binary sequence instead of a `str`. The `%` operator does not work with binary sequences in Python 3.0 to 3.4, but should be supported in version 3.5 according to [PEP 461 — Adding % formatting to bytes and bytearray](#).

Binary sequences have a class method that `str` doesn't have: `fromhex`, which builds a binary sequence by parsing pairs of hex digits optionally separated by spaces:

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

The other ways of building bytes or bytearray instances are calling their constructors with:

- a `str` and an encoding keyword argument.
- an iterable providing items with values from 0 to 255.
- a single integer, to create a binary sequence of that size initialized with null bytes<sup>3</sup>.
- an object that implements the buffer protocol (eg. `bytes`, `bytearray`, `memoryview`, `array.array`); this copies the bytes from the source object to the newly created binary sequence.

Building a binary sequence from a buffer-like object is a low-level operation that may involve type casting. See a demonstration in [Example 4-3](#).

*Example 4-3. Initializing bytes from the raw data of an array.*

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
>>> octets = bytes(numbers) ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

- ❶ Typecode 'h' creates an array of short integers (16 bits).
- ❷ `octets` holds a copy of the bytes that make up `numbers`.
- ❸ These are the 10 bytes that represent the five short integers.

Creating a `bytes` or `bytearray` object from a any buffer-like source will always copy the bytes. In contrast, `memoryview` objects let you share memory between binary data structures. To extract structured information from binary sequences, the `struct` module is invaluable. We'll see it working along with `bytes` and `memoryview` in the next section.

3. This signature will be deprecated in Python 3.5 and removed in Python 3.6. See [PEP 467 — Minor API improvements for binary sequences](#).

## Structs and memory views

The `struct` module provides functions to parse packed bytes into a tuple of fields of different types and to perform the opposite conversion, from a tuple into packed bytes. `struct` is used with `bytes`, `bytearray` and `memoryview` objects.

As we've seen in “[Memory views](#)” on page 51, `memoryview` class does not let you create or store byte sequences, but provides shared memory access to slices of data from other binary sequences, packed arrays and buffers such as PIL images<sup>4</sup>, without copying the bytes.

**Example 4-4** shows the use of `memoryview` and `struct` together to extract the width and height of a GIF image.

*Example 4-4. Using `memoryview` and `struct` to inspect a GIF image header.*

```
>>> import struct
>>> fmt = '<3s3sHH' # ❶
>>> with open('filter.gif', 'rb') as fp:
...     img = memoryview(fp.read()) # ❷
...
>>> header = img[:10] # ❸
>>> bytes(header) # ❹
b'GIF89a+\x02\xe6\x00'
>>> struct.unpack(fmt, header) # ❺
(b'GIF', b'89a', 555, 230)
>>> del header # ❻
>>> del img
```

- ❶ struct format: < little-endian; 3s3s two sequences of 3 bytes; HH two 16-bit integers.
- ❷ Create `memoryview` from file contents in memory...
- ❸ ...then another `memoryview` by slicing the first one; no bytes are copied here.
- ❹ Convert to `bytes` for display only; 10 bytes are copied here.
- ❺ Unpack `memoryview` into tuple of: type, version, width and height.
- ❻ Delete references to release the memory associated with the `memoryview` instances.

Note that slicing a `memoryview` returns a new `memoryview`, without copying bytes<sup>5</sup>.

4. PIL is the Python Imaging Library, and [Pillow](#) is its most active fork.

5. Leonardo Rochaël — one of the technical reviewers — pointed out that even less byte copying would happen if I used the `mmap` module to open the image as a memory-mapped file. I will not cover `mmap` in this book, but if read and change binary files frequently, learning more about [mmap — Memory-mapped file support](#) will be very fruitful.

We will not go deeper into `memoryview` or the `struct` module in this book, but if you work with binary data, you'll find it worthwhile to study their docs: [Built-in Types » Memory Views](#) and [struct — Interpret bytes as packed binary data](#).

After this brief exploration of binary sequence types in Python, let us see how they are converted to/from strings.

## Basic encoders/decoders

The Python distribution bundles more than 100 *codecs* (encoder/decoder) for text to byte conversion and vice-versa. Each codec has a name, like `'utf_8'`, and often aliases, such as `'utf8'`, `'utf-8'` and `'U8'`, which you can use as the `encoding` argument in functions like `open()`, `str.encode()`, `bytes.decode()` and so on.

*Example 4-5. The string “El Niño” encoded with three codecs producing very different byte sequences.*

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xff1o'
utf_8   b'El Ni\xc3\x10'
utf_16  b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\x10\x00o\x00'
```

**Figure 4-1** demonstrates a variety of codecs generating bytes from characters like the letter “A” through the G-clef musical symbol. Note that the last three encodings are variable-length, multi-byte encodings.

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
€	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
Г	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
♯	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

*Figure 4-1. Twelve characters, their code points and their byte representation (in hex) in seven different encodings. \* means the character cannot be represented in that encoding.*

All those stars \* in [Figure 4-1](#) make clear that some encodings, like ASCII and even the multi-byte GB2312, cannot represent every Unicode character. The UTF encodings, however, are designed to handle every Unicode code point.

The encodings shown in [Figure 4-1](#) were chosen as a representative sample:

**latin1 a.k.a. iso8859\_1**

Important because it is the basis for other encodings, such as cp1252 and Unicode itself (note how the latin1 byte values appear in the cp1252 bytes and even in the code points).

**cp1252**

A latin1 superset by Microsoft, adding useful symbols like curly quotes and the € (euro); some Windows apps call it “ANSI”, but it was never a real ANSI standard.

**cp437**

The original character set of the IBM PC, with box drawing characters. Incompatible with latin1, which appeared later.

**gb2312**

Legacy standard to encode the simplified Chinese ideographs used in mainland China; one of several widely deployed multi-byte encodings for Asian languages.

**utf-8**

The most common 8-bit encoding on the Web, by far<sup>6</sup>; backward-compatible with ASCII (pure ASCII text is valid UTF-8).

**utf-16le**

One form of the UTF-16 16-bit encoding scheme; all UTF-16 encodings support code points beyond U+FFFF through escape sequences called “surrogate pairs”.



UTF-16 superseded the original 16-bit Unicode 1.0 encoding — UCS-2 — way back in 1996. UCS-2 is still deployed in many systems, but it only supports code points up to U+FFFF. As of Unicode 6.3, more than 50% of the allocated code points are above U+10000, including the increasingly popular emoji pictographs.

After this overview of common encodings, we now move to handling issues in encoding and decoding operations.

6. As of September, 2014, [W3Techs: Usage of character encodings for websites](#) claims that 81.4% of sites use UTF-8, while [Built With: Encoding Usage Statistics](#) estimates 79.4%.



# Understanding encode/decode problems

Although there is a generic `UnicodeError` exception, almost always the error reported is more specific: either an `UnicodeEncodeError`, when converting `str` to binary sequences or an `UnicodeDecodeError` when reading binary sequences into `str`. Loading Python modules may also generate `SyntaxError` when the source encoding is unexpected. We'll show how to handle all of these errors in the next sections.



The first thing to note when you get a Unicode error is the exact type of the exception. Is it an `UnicodeEncodeError`, an `UnicodeDecodeError` or some other error (eg. `SyntaxError`) that mentions an encoding problem? To solve the problem you have to understand it first.

## Coping with `UnicodeEncodeError`

Most non-UTF codecs handle only a small subset of the Unicode characters. When converting text to bytes, if a character is not defined in the target encoding, `UnicodeEncodeError` will be raised, unless special handling is provided by passing an `errors` argument to the encoding method or function. The behavior of the error handlers is shown in [Example 4-6](#).

*Example 4-6. Encoding to bytes: success and error handling*

```
>>> city = 'São Paulo'
>>> city.encode('utf_8') ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1') ❷
b'S\xe3o Paulo'
>>> city.encode('cp437') ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../lib/python3.4/encodings/cp437.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore') ❹
b'So Paulo'
>>> city.encode('cp437', errors='replace') ❺
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace') ❻
b'S&#227;o Paulo'
```

- ❶ The `'utf_?'` encodings handle any `str`.
- ❷ `'iso8859_1'` also works for the `'São Paulo'` `str`.

- ③ 'cp437' can't encode the 'ä' (“a” with tilde). The default error handler — 'strict' — raises `UnicodeEncodeError`.
- ④ The `error='ignore'` handler silently skips characters that cannot be encoded; this is usually a very bad idea.
- ⑤ When encoding, `error='replace'` substitutes unencodable characters with '?'; data is lost, but users will know something is amiss.
- ⑥ 'xmlcharrefreplace' replaces unencodable characters with a XML entity.



The codecs error handling is extensible. You may register extra strings for the `errors` argument by passing a name and an error handling function to the `codecs.register_error` function. See the [codecs.register\\_error](#) documentation.

## Coping with `UnicodeDecodeError`

Not every byte holds a valid ASCII character, and not every byte sequence is valid UTF-8 or UTF-16, therefore when you assume one of these encodings while converting a binary sequence to text, you will get a `UnicodeDecodeError` if unexpected bytes are found.

On the other hand, many legacy 8-bit encodings like 'cp1252', 'iso8859\_1', 'koi8\_r' are able to decode any stream of bytes, including random noise, without generating errors. Therefore, if your program assumes the wrong 8-bit encoding, it will silently decode garbage.



Garbled characters are known as gremlins or mojibake (???? - Japanese for “transformed text”)⁷.

*Example 4-7. Decoding from `str` to bytes: success and error handling*

```
>>> octets = b'Montr\x9a\x9b' ①
>>> octets.decode('cp1252') ②
'Montréal'
>>> octets.decode('iso8859_7') ③
'Montréal'
>>> octets.decode('koi8_r') ④
'MontrIal'
```

7. Ironically, the rendered PDF of this book currently shows gremlins where the Japanese characters for “mojibake” should be. The English language Wikipedia article for [Mojibake](#) shows the word in Japanese, if you're curious.

```
>>> octets.decode('utf_8') ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace') ❹
'Montr  al'
```

- ❶ These bytes are the characters for “Montréal” encoded as latin1; '\xe9' is the byte for “  ”.
- ❷ Decoding with 'cp1252' (Windows 1252) works because it is a proper superset of latin1.
- ❸ ISO-8859-7 is intended for Greek, so the '\xe9' byte is misinterpreted, and no error is issued.
- ❹ KOI8-R is for Russian. Now '\xe9' stands for the Cyrillic letter “  ”.
- ❺ The 'utf\_8' codec detects that octets is not valid UTF-8, and raises Unicode DecodeError
- ❻ Using 'replace' error handling, the \xe9 is replaced by “  ” (code point U+FFFD), the official Unicode REPLACEMENT CHARACTER intended to represent unknown characters.

## SyntaxError when loading modules with unexpected encoding

UTF-8 is the default source encoding for Python 3, just as ASCII was the default for Python 2 (starting with 2.5). If you load a .py module containing non-UTF-8 data and no encoding declaration, you get a message like this:

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line
 1, but no encoding declared; see http://python.org/dev/peps/pep-0263/
for details
```

Because UTF-8 is widely deployed in GNU/Linux and OSX systems, a likely scenario is opening a .py file created on Windows with cp1252. Note that this error happens even in Python for Windows, because the default encoding for Python 3 is UTF-8 across all platforms.

To fix this problem, add a magic coding comment at the top of the file:

*Example 4-8. 'ola.py': “Hello, World!” in Portuguese.*

```
# coding: cp1252
```

```
print('Ol  , Mundo!')
```



Now that Python 3 source code is no longer limited to ASCII and defaults to the excellent UTF-8 encoding, the best “fix” for source code in legacy encodings like 'cp1252' is to convert them to UTF-8 already, and not bother with the coding comments. If your editor does not support UTF-8, it's time to switch.

## Non-ASCII names in source code: should you use them?

Python 3 allows non-ASCII identifiers in source code:

```
>>> ação = 'PBR' # ação = stock
>>> ε = 10**-6   # ε = epsilon
```

Some people dislike the idea. The most common argument to stick with ASCII identifiers is to make it easy for everyone to read and edit code. That argument misses the point: you want your source code to be readable and editable by its intended audience, and that may not be “everyone”. If the code belongs to a multi-national corporation or is Open Source and you want contributors from around the World, the identifiers should be in English, and then all you need is ASCII.

But if you are a teacher in Brazil, your students will find it easier to read code that uses Portuguese variable and function names, correctly spelled. And they will have no difficulty typing the cedillas and accented vowels on their localized keyboards.

Now that Python can parse Unicode names and UTF-8 is the default source encoding, I see no point in coding identifiers in Portuguese without accents, as we used to do in Python 2 out of necessity — unless you need the code to run on Python 2 also. If the names are in Portuguese, leaving out the accents won't make the code more readable to anyone.

This is my point of view as a Portuguese-speaking Brazilian, but I believe it applies across borders and cultures: choose the human language that makes the code easier to read by the team, then use the characters needed for correct spelling.

Suppose you have a text file, be it source code or poetry, but you don't know its encoding. How to detect the actual encoding? The next section answers that with a library recommendation.

## How to discover the encoding of a byte sequence

Short answer: you can't. You must be told.

Some communication protocols and file formats, like HTTP and XML, contain headers that explicitly tell us how the content is encoded. You can be sure that some byte streams are not ASCII because they contain byte values over 127, and the way UTF-8 and UTF-16

are built also limits the possible byte sequences. But even then, you can never be 100% positive that a binary file is ASCII or UTF-8 just because certain bit patterns are not there.

However, considering that human languages also have their rules and restrictions, once you assume that a stream of bytes is human *plain text* it may be possible to sniff out its encoding using heuristics and statistics. For example, if b'\x00' bytes are common, it is probably a 16 or 32-bit encoding, and not an 8-bit scheme, because null characters in plain text are bugs; when the byte sequence b'\x20\x00' appears often, it is likely to be the space character (U+0020) in a UTF-16LE encoding, rather than the obscure U+2000 EN QUAD character — whatever that is.

That is how the package **Chardet — The Universal Character Encoding Detector** works to identify one of 30 supported encodings. Chardet is a Python library that you can use in your programs, but also includes a command-line utility, `chardetect`. Here is what it reports on the source file for this chapter:

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

Although binary sequences of encoded text usually don't carry explicit hints of their encoding, the UTF formats may prepend a byte order mark to the textual content. That is explained next.

## BOM: a useful gremlin

In **Example 4-5** you may have noticed a couple of extra bytes at the beginning of an UTF-16 encoded sequence. Here they are again:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xfa\x00o\x00'
```

The bytes are: b'\xff\xfe'. That is a *BOM* — byte-order mark — denoting the “little-endian” byte ordering of the Intel CPU where the encoding was performed.

On a little-endian machine, for each code point the least significant byte comes first: the letter 'E', code point U+0045 (decimal 69), is encoded in byte offsets 2 and 3 as 69 and 0:

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

On a big-endian CPU, the encoding would be reversed, 'E' would be encoded as 0 and 69.

To avoid confusion, the UTF-16 encoding prepends the text to be encoded with the special character ZERO WIDTH NO-BREAK SPACE (U+FEFF) which is invisible. On a little-endian system, that is encoded as b'\xff\xfe' (decimal 255, 254). Because, by design,

there is no U+FFFE character, the byte sequence `b'\xff\xfe'` must mean the ZERO WIDTH NO-BREAK SPACE on a little-endian encoding, so the codec knows which byte ordering to use.

There is a variant of UTF-16 — UTF-16LE — that is explicitly little endian, and another one explicitly big-endian, UTF-16BE. If you use them, a BOM is not generated:

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

If present, the BOM is supposed to be filtered by the UTF-16 codec, so that you only get the actual text contents of the file without the leading ZERO WIDTH NO-BREAK SPACE. The standard says that if a file is UTF-16 and has no BOM, it should be assumed to be UTF-16BE (big-endian). However, the Intel x86 architecture is little-endian, so there is plenty of little-endian UTF-16 with no BOM in the wild.

This whole issue of endianness only affects encodings that use words of more than one byte, like UTF-16 and UTF-32. One big advantage of UTF-8 is that it produces the same byte sequence regardless of machine endianness, so no BOM is needed. Nevertheless, some Windows applications (notably Notepad) add the BOM to UTF-8 files anyway — and Excel depends on the BOM to detect an UTF-8 file, otherwise it assumes the content is encoded with a Windows codepage. The character U+FEFF encoded in UTF-8 is the three-byte sequence `b'\xef\xbb\xbf'`. So if a file starts with those three bytes, it is likely to be a UTF-8 file with a BOM. However, Python does not automatically assume a file is UTF-8 just because it starts with `b'\xef\xbb\xbf'`.

We now move to handling text files in Python 3.

## Handling text files

## The Unicode sandwich



bytes → str

Decode bytes on input,

100% str

process text only,

str → bytes

encode text on output.

Figure 4-2. Unicode sandwich: current best practice for text processing.

The best practice for handling text is the “Unicode sandwich” (Figure 4-2)<sup>8</sup>. This means that bytes should be decoded to `str` as early as possible on input, e.g. when opening a file for reading. The “meat” of the sandwich is the business logic of your program, where text handling is done exclusively on `str` objects. You should never be encoding or decoding in the middle of other processing. On output, the `str` are encoded to bytes as late as possible. Most Web frameworks work like that, and we rarely touch bytes when using them. In Django, for example, your views should output Unicode `str`; Django itself takes care of encoding the response to bytes, using UTF-8 by default.

Python 3 makes it easier to follow the advice of the Unicode sandwich, because the open built-in does the necessary decoding when reading and encoding when writing files in text mode, so all you get from `my_file.read()` and pass to `my_file.write(text)` are `str` objects<sup>9</sup>.

Therefore, using text files is simple. But if you rely on default encodings you will get bitten.

Consider the console session in Example 4-9. Can you spot the bug?

*Example 4-9. A platform encoding issue. If you try this on your machine, you may or may not see the problem.*

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

8. I first saw the term “Unicode sandwich” in Ned Batchelder’s excellent *Pragmatic Unicode* talk at US PyCon 2012

9. Python 2.6 or 2.7 users have to use `io.open()` to get automatic decoding/encoding when reading/writing.

The bug: I specified UTF-8 encoding when writing the file but failed to do so when reading it, so Python assumed the system default encoding — Windows 1252 — and the trailing bytes in the file were decoded as characters 'Ã©' instead of 'é'.

I ran [Example 4-9](#) on a Windows 7 machine. The same statements running on recent GNU/Linux or Mac OSX work perfectly well because their default encoding is UTF-8, giving the false impression that everything is fine. If the encoding argument was omitted when opening the file to write, the locale default encoding would be used, and we'd read the file correctly using the same encoding. But then this script would generate files with different byte contents depending on the platform or even depending on locale settings in the same platform, creating compatibility problems.



Code that has to run on multiple machines or on multiple occasions should never depend on encoding defaults. Always pass an explicit `encoding=` argument when opening text files, because the default may change from one machine to the next, or from one day to the next.

A curious detail in [Example 4-9](#) is that the `write` function in the first statement reports that 4 characters were written, but in the next line 5 characters are read. [Example 4-10](#) is an extended version of [Example 4-9](#), explaining that and other details.

*Example 4-10. Closer inspection of [Example 4-9](#) running on Windows reveals the bug and how to fix it.*

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp ①
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café')
4 ②
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size
5 ③
>>> fp2 = open('cafe.txt')
>>> fp2 ④
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ⑤
'cp1252'
>>> fp2.read()
'cafÃ©' ⑥
>>> fp3 = open('cafe.txt', encoding='utf_8') ⑦
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read()
'café' ⑧
>>> fp4 = open('cafe.txt', 'rb') ⑨
>>> fp4
```



```
<_io.BufferedReader name='cafe.txt'> ❩
>>> fp4.read() ❪
b'caf\xc3\xa9'
```

- ❶ By default, `open` operates in text mode and returns a `TextIOWrapper` object.
- ❷ The `write` method on a `TextIOWrapper` returns the number of Unicode characters written.
- ❸ `os.stat` reports that the file holds 5 bytes; UTF-8 encodes 'é' as two bytes, 0xc3 and 0xa9.
- ❹ Opening a text file with no explicit encoding returns a `TextIOWrapper` with the encoding set to a default from the locale.
- ❺ A `TextIOWrapper` object has an `encoding` attribute that you can inspect: `cp1252` in this case.
- ❻ In the Windows `cp1252` encoding, the byte 0xc3 is an “Ä” (A with tilde) and 0xa9 is the copyright sign.
- ❼ Opening the same file with the correct encoding.
- ❽ The expected result: the same 4 Unicode characters for 'café'.
- ❾ The 'rb' flag opens a file for reading in binary mode.
- ❿ The returned object is a `BufferedReader` and not a `TextIOWrapper`.
- ⓫ Reading that returns bytes, as expected.



Do not open text files in binary mode unless you need to analyze the file contents to determine the encoding — even then, you should be using `Chardet` instead of reinventing the wheel (see “[How to discover the encoding of a byte sequence](#)” on page 108). Ordinary code should only use binary mode to open binary files, like raster images.

The problem in [Example 4-10](#) has to do with relying on a default setting while opening a text file. There are several sources for such defaults, as the next section shows.

## Encoding defaults: a madhouse

Several settings affect the encoding defaults for I/O in Python. See the `default_encodings.py` script in [Example 4-11](#).

*Example 4-11. Exploring encoding defaults*

```
import sys, locale

expressions = """
    locale.getpreferredencoding()
```

```

type(my_file)
my_file.encoding
sys.stdout.isatty()
sys.stdout.encoding
sys.stdin.isatty()
sys.stdin.encoding
sys.stderr.isatty()
sys.stderr.encoding
sys.getdefaultencoding()
sys.getfilesystemencoding()
"""

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(expression.rjust(30), '->', repr(value))

```

The output of [Example 4-11](#) on GNU/Linux (Ubuntu 14.04) and OSX (Mavericks 10.9) is identical, showing that UTF-8 is used everywhere in these systems:

```

$ python3 default_encodings.py
locale.getpreferredencoding() -> 'UTF-8'
type(my_file) -> <class '_io.TextIOWrapper'>
my_file.encoding -> 'UTF-8'
sys.stdout.isatty() -> True
sys.stdout.encoding -> 'UTF-8'
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'UTF-8'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'UTF-8'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'utf-8'

```

On Windows, however, the output is [Example 4-12](#).

*Example 4-12. Default encodings on Windows 7 (SP 1) cmd.exe localized for Brazil; PowerShell gives same result.*

```

Z:\>chcp ①
Página de código ativa: 850
Z:\>python default_encodings.py ②
locale.getpreferredencoding() -> 'cp1252' ③
type(my_file) -> <class '_io.TextIOWrapper'>
my_file.encoding -> 'cp1252' ④
sys.stdout.isatty() -> True ⑤
sys.stdout.encoding -> 'cp850' ⑥
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'cp850'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'cp850'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'mbcs'

```

- ❶ `chcp` shows the active codepage for the console: 850.
- ❷ Running `default_encodings.py` with output to console.
- ❸ `locale.getpreferredencoding()` is the most important setting.
- ❹ Text files use `locale.getpreferredencoding()` by default.
- ❺ The output is going to the console, so `sys.stdout.isatty()` is `True`.
- ❻ Therefore, `sys.stdout.encoding` is the same as the console encoding.

If the output is redirected to a file, like this:

```
Z:\>python default_encodings.py > encodings.log
```

Then the value of `sys.stdout.isatty()` becomes `False`, and `sys.stdout.encoding` is set by `locale.getpreferredencoding()`, 'cp1252' in that machine.

Note that there are 4 different encodings in [Example 4-12](#):

- If you omit the `encoding` argument when opening a file, the default is given by `locale.getpreferredencoding()` ('cp1252' in [Example 4-12](#)).
- The encoding of `sys.stdout/stdin/stderr` is given by the <https://docs.python.org/3/using/cmdline.html#envvar-PYTHONIOENCODING> [`PYTHONIOENCODING`] environment variable, if present, otherwise it is either inherited from the console or defined by `locale.getpreferredencoding()` if the output/input is redirected to/from a file.
- `sys.getdefaultencoding()` is used internally by Python to convert binary data to/from `str`; this happens less often in Python 3, but still happens<sup>10</sup>. Changing this setting is not supported<sup>11</sup>.
- `sys.getfilesystemencoding()` is used to encode/decode file names (not file contents). It is used when `open()` gets a `str` argument for the file name; if the file name is given as a `bytes` argument, it is passed unchanged to the OS API. The Python [Unicode HOWTO](#) says: “on Windows, Python uses the name `mbcs` to refer to whatever the currently configured encoding is.” The acronym MBCS stands for Multi Byte Character Set, which for Microsoft are the legacy variable-width encodings

10. While researching this subject I did not find a list of situations when Python 3 internally converts `bytes` to `str`. Python core developer Antoine Pitrou says on the `comp.python.devel` list that CPython internal functions that depend on such conversions “don’t get a lot of use in py3k” (<http://article.gmane.org/gmane.comp.python.devel/110036>)

11. The Python 2 `sys.setdefaultencoding` function was misused and is no longer documented in Python 3. It was intended for use by the core developers when the internal default encoding of Python was still undecided. In the same `comp.python.devel` thread, Marc-André Lemburg states that the `sys.setdefaultencoding` must never be called by user code and the only values supported by CPython are 'ascii' in Python 2 and 'utf-8' in Python 3 (see <http://article.gmane.org/gmane.comp.python.devel/109916>).

like `gb2312` or `Shift_JIS`, but not UTF-8<sup>footnote:</sup>[On this topic, a useful answer on StackOverflow is [Difference between MBCS and UTF-8 on Windows](#)..



On GNU/Linux and OSX all of these encodings are set to UTF-8 by default, and have been for several years, so I/O handles all Unicode characters. On Windows, not only are different encodings used in the same system, but they are usually codepages like `'cp850'` or `'cp1252'` that support only ASCII with 127 additional characters that are not the same from one encoding to the other. Therefore, Windows users are far more likely to face encoding errors unless they are extra careful.

To summarize, the most important encoding setting is that returned by `locale.getpreferredencoding()`: it is the default for opening text files and for `sys.stdout/stdin/stderr` when they are redirected to files. However, the [documentation](#) reads (in part):

```
locale.getpreferredencoding(do_setlocale=True)
```

Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess. [...]

Therefore, the best advice about encoding defaults is: do not rely on them.

If you follow the advice of the Unicode sandwich and always are explicit about the encodings in your programs, you will avoid a lot of pain. Unfortunately Unicode is painful even if you get your bytes correctly converted to `str`. The next two sections cover subjects that are simple in ASCII-land, but get quite complex on planet Unicode: text normalization — i.e. converting text to a uniform representation for comparisons — and sorting.

## Normalizing Unicode for saner comparisons

String comparisons are complicated by the fact that Unicode has combining characters: diacritics and other marks that attach to the preceding character, appearing as one when printed.

For example, the word “café” may be composed in two ways, using 4 or 5 code points, but the result looks exactly the same:

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

The code point U+0301 is the COMBINING ACUTE ACCENT. Using it after “é” renders “ë”. In the Unicode standard, sequences like 'é' and 'e\u0301' are called “canonical equivalents”, and applications are supposed to treat them as the same. But Python sees two different sequences of code points, and considers them not equal.

The solution is to use Unicode normalization, provided by the `unicodedata.normalize` function. The first argument to that function is one of four strings: 'NFC', 'NFD', 'NFKC' and 'NFKD'. Let's start with the first two.

NFC (Normalization Form C) composes the code points to produce the shortest equivalent string, while NFD decomposes, expanding composed characters into base characters and separate combining characters. Both of these normalizations make comparisons work as expected:

```
>>> from unicodedata import normalize
>>> s1 = 'café' # composed "e" with acute accent
>>> s2 = 'cafe\u0301' # decomposed "e" and acute accent
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

Western keyboards usually generate composed characters, so text typed by users will be in NFC by default. But to be safe it may be good to sanitize strings with `normalize('NFC', user_text)` before saving. NFC is also the normalization form recommended by the W3C in [Character Model for the World Wide Web: String Matching and Searching](#).

Some single characters are normalized by NFC into another single character. The symbol for the ohm  $\Omega$  unit of electrical resistance is normalized to the Greek uppercase omega. They are visually identical, but they compare unequal so it is essential to normalize to avoid surprises:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

The letter K in the acronym for the other two normalization forms — NFKC and NFKD — stands for “compatibility”. These are stronger forms of normalization, affecting the so called “compatibility characters”. Although one goal of Unicode is to have a single “canonical” code point for each character, some characters appear more than once for compatibility with preexisting standards. For example, the micro sign, 'μ' (U+00B5) was added to Unicode to support round-trip conversion to `latin1`, even though the same character is part of the Greek alphabet with code point is U+03BC (GREEK SMALL LETTER MU). So, the micro sign is considered a “compatibility character”.

In the NFKC and NFKD forms, each compatibility character is replaced by a “compatibility decomposition” of one or more characters that are considered a “preferred” representation, even if there is some formatting loss — ideally, the formatting should be the responsibility of external markup, not part of Unicode. To exemplify, the compatibility decomposition of the one half fraction '½' (U+00BD) is the sequence of three characters '1/2', and the compatibility decomposition of the micro sign 'μ' (U+00B5) is the lowercase mu 'μ' (U+03BC)<sup>12</sup>.

Here is how the NFKC works in practice:

```
>>> from unicodedata import normalize, name
>>> half = '½'
>>> normalize('NFKC', half)
'1/2'
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'μ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('μ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

Although '1/2' is a reasonable substitute for '½', and the micro sign is really a lowercase Greek mu, converting '4²' to '42' changes the meaning<sup>13</sup>. An application could store '4²' as '4<sup>2</sup>', but the `normalize` function knows nothing about formatting. Therefore, NFKC or NFKD may lose or distort information, but they can produce

12. Curiously, the micro sign is considered a “compatibility character” but the ohm symbol is not. The end result is that NFC doesn’t touch the micro sign but changes the ohm symbol to capital omega, while NFKC and NFKD change both the ohm and the micro into other characters.

13. This could lead some to believe that 16 is The Answer to the Ultimate Question of Life, The Universe, and Everything.

convenient intermediate representations for searching and indexing: users may be pleased that a search for '1/2 inch' also finds documents containing '½ inch'.



NFKC and NFKD normalization should be applied with care and only in special cases — e.g. search and indexing — and not for permanent storage, as these transformations cause data loss.

When preparing text for searching or indexing, another operation is useful: case folding, our next subject.

## Case folding

Case folding is essentially converting all text to lowercase, with some additional transformations. It is supported by the `str.casefold()` method (new in Python 3.3).

For any string `s` containing only `latin1` characters, `s.casefold()` produces the same result as `s.lower()`, with only two exceptions: the micro sign 'μ' is changed to the Greek lower case mu (which looks the same in most fonts) and the German Eszett or “sharp s” (ß) becomes “ss”.

```
>>> micro = 'μ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('μ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

As of Python 3.4 there are 116 code points for which `str.casefold()` and `str.lower()` return different results. That's 0.11% of a total of 110,122 named characters in Unicode 6.3.

As usual with anything related to Unicode, case folding is a complicated issue with plenty of linguistic special cases, but the Python core team made an effort to provide a solution that hopefully works for most users.

In the next couple of sections, we'll put our normalization knowledge to use developing utility functions.

## Utility functions for normalized text matching

As we've seen, NFC and NFD are safe to use and allow sensible comparisons between Unicode strings. NFC is the best normalized form for most applications. `str.casefold()` is the way to go for case-insensitive comparisons.

If you work with text in many languages, a pair of functions like `nfc_equal` and `fold_equal` in [Example 4-13](#) are useful additions to your toolbox.

*Example 4-13. `normeq.py`: normalized Unicode string comparison.*

```
"""
Utility functions for normalized Unicode string comparison.
```

```
Using Normal Form C, case sensitive:
```

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1 == s2
False
>>> nfc_equal(s1, s2)
True
>>> nfc_equal('A', 'a')
False
```

```
Using Normal Form C with case folding:
```

```
>>> s3 = 'Straße'
>>> s4 = 'strasse'
>>> s3 == s4
False
>>> nfc_equal(s3, s4)
False
>>> fold_equal(s3, s4)
True
>>> fold_equal(s1, s2)
True
>>> fold_equal('A', 'a')
True
```

```
"""
```

```
from unicodedata import normalize
```

```
def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```



Beyond Unicode normalization and case folding — both part of the Unicode standard — sometimes it makes sense to apply deeper transformations, like changing 'café' into 'cafe'. We'll see when and how in the next section.

## Extreme “normalization”: taking out diacritics

The Google Search secret sauce involves many tricks, but one of them apparently is ignoring diacritics (e.g. accents, cedillas etc.), at least in some contexts. Removing diacritics is not a proper form of normalization because it often changes the meaning of words and may produce false positives when searching. But it helps coping with some facts of life: people sometimes are lazy or ignorant about the correct use of diacritics, and spelling rules change over time, meaning that accents come and go in living languages.

Outside of searching, getting rid of diacritics also makes for more readable URLs, at least in Latin based languages. Take a look at the URL for the English language Wikipedia article about the city of São Paulo:

```
http://en.wikipedia.org/wiki/S%C3%A3o_Paulo
```

The %C3%A3 part is the URL-escaped, UTF-8 rendering of the single letter “ã” (“a” with tilde). The following is much friendlier, even if it is not the right spelling:

```
http://en.wikipedia.org/wiki/Sao_Paulo
```

To remove all diacritics from a `str`, you can use a function like [Example 4-14](#).

*Example 4-14. Function to remove all combining marks (module `sanitize.py`)*

```
import unicodedata
import string

def shave_marks(txt):
    """Remove all diacritic marks"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c)) ❷
    return unicodedata.normalize('NFC', shaved) ❸
```

- ❶ Decompose all characters into base characters and combining marks.
- ❷ Filter out all combining marks.
- ❸ Recompose all characters.

[Example 4-15](#) shows a couple of uses of `shave_marks`.

Example 4-15. Two examples using `shave_marks` from Example 4-14.

```
>>> order = '"Herr Voß: • ½ cup of Ætker™ caffè latte • bowl of açai."'
>>> shave_marks(order)
'"Herr Voß: • ½ cup of Ætker™ caffè latte • bowl of açai."' ❶
>>> Greek = 'Ζέφυρος, Ζέφиро'
>>> shave_marks(Greek)
'Ζεφυρος, Zefiro' ❷
```

- ❶ Only the letters “è”, “ç” and “ï” were replaced.
- ❷ Both “έ” and “ε” were replaced.

The function `shave_marks` from Example 4-14 works all right, but maybe it goes too far. Often the reason to remove diacritics is to change Latin text to pure ASCII, but `shave_marks` also changes non-Latin characters — like Greek letters — which will never become ASCII just by losing their accents. So it makes sense to analyze each base character and to remove attached marks only if the base character is a letter from the Latin alphabet. This is what Example 4-16 does.

Example 4-16. Function to remove combining marks from Latin characters. `import` statements are omitted as this is part of the `sanitize.py` module from Example 4-14.

```
def shave_marks_latin(txt):
    """Remove all diacritic marks from Latin base characters"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    latin_base = False
    keepers = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base: ❷
            continue # ignore diacritic on Latin base char
        keepers.append(c) ❸
        # if it isn't combining char, it's a new base char
        if not unicodedata.combining(c): ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(keepers)
    return unicodedata.normalize('NFC', shaved) ❺
```

- ❶ Decompose all characters into base characters and combining marks.
- ❷ Skip over combining marks when base character is Latin.
- ❸ Otherwise, keep current character.
- ❹ Detect new base character and determine if it's Latin.
- ❺ Recompose all characters.

An even more radical step would be to replace common symbols in Western texts, like curly quotes, em-dashes, bullets etc. into ASCII equivalents. This is what the function `asciize` does in Example 4-17.

Example 4-17. Transform some Western typographical symbols into ASCII. This snippet is also part of `sanitize.py` from [Example 4-14](#).

```
single_map = str.maketrans(" ", f„†^<'>“”•—~>""", ❶
                        """"'f"*^<'""'-...~>""")

multi_map = str.maketrans({ ❷
    '€': '<euro>',
    '…': '...',
    'Œ': 'OE',
    '™': '(TM)',
    'œ': 'oe',
    '‰': '<per mille>',
    '‡': '**',
})

multi_map.update(single_map) ❸

def dewinize(txt):
    """Replace Win1252 symbols with ASCII chars or sequences"""
    return txt.translate(multi_map) ❹

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt)) ❺
    no_marks = no_marks.replace('ß', 'ss') ❻
    return unicodedata.normalize('NFKC', no_marks) ❼
```

- ❶ Build mapping table for char to char replacement.
- ❷ Build mapping table for char to string replacement.
- ❸ Merge mapping tables.
- ❹ `dewinize` does not affect ASCII or latin1 text, only the Microsoft additions in to latin1 in cp1252.
- ❺ Apply `dewinize` and remove diacritical marks.
- ❻ Replace the Eszett with “ss” (we are not using case fold here because we want to preserve the case).
- ❼ Apply NFKC normalization to compose characters with their compatibility code points.

[Example 4-18](#) shows `asciize` in use.

Example 4-18. Two examples using `asciize` from [Example 4-17](#).

```
>>> order = "Herr Voß: • ½ cup of Œtker™ caffè latte • bowl of açai."
>>> dewinize(order)
'Herr Voß: - ½ cup of Œtker(TM) caffè latte - bowl of açai.' ❶
```

```
>>> asciize(order)
'Herr Voss: - 1/2 cup of OEtker(TM) caffe latte - bowl of acai.'" ❷
```

- ❶ dewinize replaces curly quotes, bullets, and <sup>™</sup> (trade mark symbol).
- ❷ asciize applies dewinize, drops diacritics and replaces the 'ß'.



Different languages have their own rules for removing diacritics. For example, Germans change the 'ü' into 'ue'. Our `asciize` function is not as refined, so it may or not be suitable for your language. It works acceptably for Portuguese, though.

To summarize, the functions in `sanitize.py` go way beyond standard normalization and perform deep surgery on the text, with a good chance of changing its meaning. Only you can decide whether to go so far, knowing the target language, your users and how the transformed text will be used.

This wraps up our discussion of normalizing Unicode text.

The next Unicode matter to sort out is... sorting.

## Sorting Unicode text

Python sorts sequences of any type by comparing the items in each sequence one by one. For strings, this means comparing the code points. Unfortunately, this produces unacceptable results for anyone who uses non-ASCII characters.

Consider sorting a list of fruits grown in Brazil:

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açai', 'caju', 'cajá']
```

Sorting rules vary for different locales, but in Portuguese and many languages that use the Latin alphabet, accents and cedillas rarely make a difference when sorting<sup>14</sup>. So “cajá” is sorted as “caja”, and must come before “caju”.

The sorted `fruits` list should be:

```
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

The standard way to sort non-ASCII text in Python is to use the `locale.strxfrm` function which, according to the [locale module docs](#), “transforms a string to one that can be used in locale-aware comparisons”.

14. Diacritics affect sorting only in the rare case when they are the only difference between two words — in that case the word with a diacritic is sorted after the plain word.

To enable `locale.strxfrm` you must first set a suitable locale for your application, and pray that the OS supports it. On GNU/Linux (Ubuntu 14.04) with the `pt_BR` locale, the sequence of commands in [Example 4-19](#) works:

*Example 4-19. Using the `locale.strxfrm` function as sort key.*

```
>>> import locale
>>> locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
'pt_BR.UTF-8'
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=locale.strxfrm)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

So you need to call `setlocale(LC_COLLATE, «your_locale»)` before using `locale.strxfrm` as the key when sorting.

There are a few caveats, though:

- Because locale settings are global, calling `setlocale` in a library is not recommended. Your application or framework should set the locale when the process starts, and should not change it afterwards.
- The locale must be installed on the OS, otherwise `setlocale` raises a `locale.Error: unsupported locale setting` exception.
- You must know how to spell the locale name. They are pretty much standardized in the Unix derivatives as `'language_code.encoding'` but on Windows the syntax is more complicated: `Language Name-Language Variant_Region Name.code page>`. Note that the Language Name, Language Variant and Region Name parts can have spaces inside them, but the parts after the first are prefixed with special different characters: an hyphen, an underline character and a dot. All parts seem to be optional except the language name. For example: `English_United States.850` means Language Name “English”, region “United States” and codepage “850”. The language and region names Windows understands are listed in the MSDN article [Language Identifier Constants and Strings](#), while [Code Page Identifiers](#) lists the numbers for the last part<sup>15</sup>.
- The locale must be correctly implemented by the makers of the OS. I was successful on Ubuntu 14.04, but not on OSX (Mavericks 10.9). On two different Macs, the call `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` returns the string `'pt_BR.UTF-8'` with no complaints. But `sorted(fruits, key=locale.strxfrm)` produced the same

15. Thanks to Leonardo Rochaël who went beyond his duties as tech reviewer and researched these Windows details, even though he is a GNU/Linux user himself.

incorrect result as `sorted(fruits)` did. I also tried the `fr_FR`, `es_ES` and `de_DE` locales on OSX, but `locale.strxfrm` never did its job<sup>16</sup>.

So the standard library solution to internationalized sorting works, but seems to be well supported only on GNU/Linux (perhaps also on Windows, if you are an expert). Even then, it depends on locale settings, creating deployment headaches.

Fortunately there is a simpler solution: the PyUCA library, available on *PyPI*.

## Sorting with the Unicode Collation Algorithm

James Tauber, prolific Django contributor, must have felt the pain and created **PyUCA**, a pure-Python implementation of UCA — the Unicode Collation Algorithm. **Example 4-20** shows how easy it is to use.

*Example 4-20. Using the `pyuca.Collator.sort_key` method.*

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

This is friendly and just works. I tested it on GNU/Linux, OSX and Windows. Only Python 3.X is supported at this time.

PyUCA does not take the locale into account. If you need to customize the sorting, you can provide the path to a custom collation table to the `Collator()` constructor. Out of the box, it uses **`allkeys.txt`** which is bundled with the project. That's just a copy of the **Default Unicode Collation Element Table** from Unicode 6.3.0.

By the way, that table is one of the many that comprise the Unicode database, our next subject.

## The Unicode database

The Unicode standard provides an entire database — in the form of numerous structured text files — that includes not only the table mapping code points to character names, but also lot of metadata about the individual characters and how they are related. For example, the Unicode database records whether a character is printable, is a letter, is a decimal digit or is some other numeric symbol. That's how the `str` methods `is`

16. Again, I could not find a solution but did find other people reporting the same problem. Alex Martelli, one of the tech reviewers, had no problem using `setlocale` and `locale.strxfrm` on his Mac with OSX 10.9. In summary: your mileage may vary.

dentifier, isprintable, isdecimal and isnumeric work. `str.casefold` also uses information from a Unicode table.

The `unicodedata` module has functions that return character metadata, for instance, its official name in the standard, whether it is a combining character (e.g. diacritic like a combining tilde) and the numeric value of the symbol for humans (not its code point).

**Example 4-21** shows the use of `unicodedata.name()` `unicodedata.numeric()` along with the `.isdecimal()` and `.isnumeric()` methods of `str`.

*Example 4-21. Demo of Unicode database numerical character metadata. Callouts describe each column in the output.*

```
import unicodedata
import re

re_digit = re.compile(r'\d')

sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print('U+%04x' % ord(char),           ❶
          char.center(6),                 ❷
          're_dig' if re_digit.match(char) else '-', ❸
          'isdig' if char.isdigit() else '-', ❹
          'isnum' if char.isnumeric() else '-', ❺
          format(unicodedata.numeric(char), '5.2f'), ❻
          unicodedata.name(char),          ❼
          sep='\t')
```

- ❶ Code point in U+0000 format.
- ❷ Character centralized in a str of length 6.
- ❸ Show `re_dig` if character matches the `r'\d'` regex.
- ❹ Show `isdig` if `char.isdigit()` is True.
- ❺ Show `isnum` if `char.isnumeric()` is True.
- ❻ Numeric value formatted with width 5 and 2 decimal places.
- ❼ Unicode character name.

Running **Example 4-21** you get **Figure 4-3**.

```
$ python3 numerics_demo.py
U+0031  1    re_dig isdig isnum  1.00  DIGIT ONE
U+00bc  ¼    -      -      isnum  0.25  VULGAR FRACTION ONE QUARTER
U+00b2  ²    -      isdig isnum  2.00  SUPERSCRIPT TWO
U+0969  ३    re_dig isdig isnum  3.00  DEVANAGARI DIGIT THREE
U+136b  ፫    -      isdig isnum  3.00  ETHIOPIC DIGIT THREE
U+216b  XII   -      -      isnum  12.00 ROMAN NUMERAL TWELVE
U+2466  ⑦    -      isdig isnum  7.00  CIRCLED DIGIT SEVEN
U+2480  ⑬    -      -      isnum  13.00 PARENTHEZIZED NUMBER THIRTEEN
U+3285  ㊦    -      -      isnum  6.00  CIRCLED IDEOGRAPH SIX
$
```

Figure 4-3. Nine numeric characters and metadata about them; *re\_dig* means the character matches the regular expression `r'\d'`.

The sixth column of Figure 4-3 is the result of calling `unicodedata.numeric(char)` on the character. It shows that Unicode knows the numeric value of symbols that represent numbers. So if you want to create a spreadsheet application that supports tamil digits or roman numerals, go for it!

Figure 4-3 shows that the regular expression `r'\d'` matches the digit “1” and the Devanagari digit three, but not some other characters that are considered digits by the `isdigit` function. The `re` module is not as savvy about Unicode as it could be. The new `regex` module available in PyPI was designed to eventually replace `re` and provides better Unicode support<sup>17</sup>. We’ll come back to the `re` module in the next section.

Throughout this chapter we’ve used several `unicodedata` functions, but there are many more we did not cover. See the standard library documentation for the [unicodedata module](#).

We will wrap up our tour of `str` versus `bytes` with a quick look at a new trend: dual mode APIs offering functions that accept `str` or `bytes` arguments with special handling depending on the type.

## Dual mode `str` and `bytes` APIs

The standard library has functions that accept `str` or `bytes` arguments and behave differently depending on the type. Some examples are in the `re` and `os` modules.

17. Although it was not better than `re` at identifying digits in this particular sample.



## str versus bytes in regular expressions

If you build a regular expression with bytes, patterns such as `\d` and `\w` only match ASCII characters; in contrast, if these patterns are given as `str`, they match Unicode digits or letters beyond ASCII. [Example 4-22](#) and [Figure 4-4](#) compare how letters, ASCII digits, superscripts and Tamil digits are matched by `str` and bytes patterns.

*Example 4-22. `ramanujan.py`: compare behavior of simple `str` and bytes regular expressions.*

```
import re
```

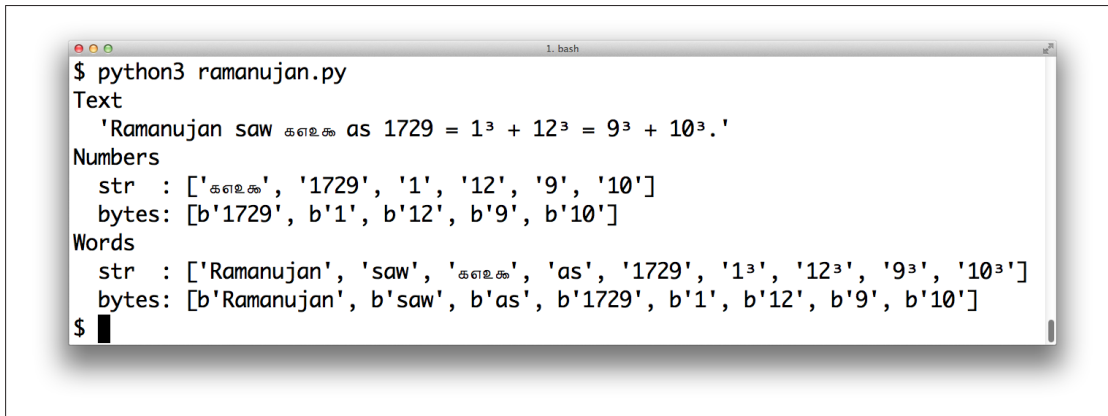
```
re_numbers_str = re.compile(r'\d+')           ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+')        ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef"  ❸
            " as 1729 = 13 + 123 = 93 + 103." )  ❹

text_bytes = text_str.encode('utf_8')          ❺

print('Text', repr(text_str), sep='\n ')
print('Numbers')
print('  str  :', re_numbers_str.findall(text_str))    ❻
print('  bytes:', re_numbers_bytes.findall(text_bytes))  ❼
print('Words')
print('  str  :', re_words_str.findall(text_str))      ❽
print('  bytes:', re_words_bytes.findall(text_bytes))  ❾
```

- ❶ The first two regular expressions are of the `str` type.
- ❷ The last two are of the bytes type.
- ❸ Unicode text to search, containing the Tamil digits for 1729 (the logical line continues until the right parenthesis token).
- ❹ This string is joined to the previous one at compile time (see [String literal concatenation](#) in the Language Reference)
- ❺ A bytes string is needed to search with the bytes regular expressions.
- ❻ The `str` pattern `r'\d+'` matches the Tamil and ASCII digits.
- ❼ The bytes pattern `rb'\d+'` matches only the ASCII bytes for digits.
- ❽ The `str` pattern `r'\w+'` matches the letters, superscripts, Tamil and ASCII digits.
- ❾ The bytes pattern `rb'\w+'` matches only the ASCII bytes for letters and digits.



```
$ python3 ramanujan.py
Text
'Ramanujan saw കളമ as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
str : ['കളമ', '1729', '1', '12', '9', '10']
bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
str : ['Ramanujan', 'saw', 'കളമ', 'as', '1729', '1³', '12³', '9³', '10³']
bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$
```

Figure 4-4. Screenshot of running `ramanujan.py` from [Example 4-22](#).

[Example 4-22](#) is a trivial example to make one point: you can use regular expressions on `str` and `bytes` but in the second case `bytes` outside of the ASCII range are treated as non-digits and non-word characters.

For `str` regular expressions there is a `re.ASCII` flag that makes `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching. See the [documentation of the `re` module](#) for full details.

Another important dual mode module is `os`.

## str versus bytes on os functions

The GNU/Linux kernel is not Unicode savvy, so in the real world you may find file names made of byte sequences that are not valid in any sensible encoding scheme, and cannot be decoded to `str`. File servers with clients using a variety of OSes are particularly prone to this problem.

To work around this issue, all `os` module functions that accept file names or path names take arguments as `str` or `bytes`. If one such function is called with a `str` argument, the argument will be automatically converted using the codec named by `sys.getfilesystemencoding()`, and the OS response will be decoded with the same codec. This is almost always what you want, in keeping with the Unicode sandwich best practice.

But if you must deal with (and perhaps fix) file names that cannot be handled in that way, you can pass `bytes` arguments to the `os` functions to get `bytes` return values. This feature lets you deal with any file or path name, no matter how many gremlins you may find. See [Example 4-23](#).

*Example 4-23. `listdir` with `str` and `bytes` arguments and results.*

```
>>> os.listdir('.') # ❶
['abc.txt', 'digits-of-n.txt']
```

```
>>> os.listdir(b'.') # ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

- ❶ The second filename is “digits-of-π.txt” (with the Greek letter pi).
- ❷ Given a byte argument, `listdir` returns filenames as bytes: `b'\xcf\x80'` is the UTF-8 encoding of the Greek letter pi).

To help with manual handling of `str` or `bytes` sequences that are file or path names, the `os` module provides special encoding and decoding functions:

`fsencode(filename)`

Encodes `filename` (can be `str` or `bytes`) to `bytes` using the codec named by `sys.getfilesystemencoding()` if `filename` is of type `str`, otherwise return the `filename` `bytes` unchanged.

`fsdecode(filename)`

Decodes `filename` (can be `str` or `bytes`) to `str` using the codec named by `sys.getfilesystemencoding()` if `filename` is of type `bytes`, otherwise return the file name `str` unchanged.

On Unix-derived platforms, these functions use the `surrogateescape` error handler (see next sidebar) to avoid choking on unexpected bytes. On Windows, the `strict` error handler is used.

## Using `surrogateescape` to deal with gremlins

A trick to deal with unexpected bytes or unknown encodings is the `surrogateescape` codec error handler described in [PEP 383 — Non-decodable Bytes in System Character Interfaces](#) introduced in Python 3.1.

The idea of this error handler is to replace each non-decodable byte with a code point in the Unicode range from `U+DC00` to `U+DCFF` that lies in the so-called “Low Surrogate Area” of the standard — a code space with no characters assigned, reserved for internal use in applications. On encoding, such code points are converted back to the byte values they replaced.

*Example 4-24. `listdir` with `str` and `bytes` arguments and results.*

```
>>> os.listdir('.') ❶
['abc.txt', 'digits-of-n.txt']
>>> os.listdir(b'.') ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
>>> pi_name_bytes = os.listdir(b'.')[1] ❸
>>> pi_name_str = pi_name_bytes.decode('ascii', 'surrogateescape') ❹
>>> pi_name_str ❺
'digits-of-\udccf\udc80.txt'
```

```
>>> pi_name_str.encode('ascii', 'surrogateescape') ❹  
b'digits-of-\xcf\x80.txt'
```

- ❶ List directory with a non-ASCII file name.
- ❷ Let's pretend we don't know the encoding and get file names as bytes.
- ❸ `pi_names_bytes` is the file name with the pi character.
- ❹ Decode it to `str` using the 'ascii' codec with 'surrogateescape'.
- ❺ Each non-ASCII byte is replaced by a surrogate code point: `'\xcf\x80'` becomes `'\udccf\udc80'`
- ❻ Encode back to ASCII bytes: each surrogate code point is replaced by the byte it replaced.

This ends our exploration of `str` and bytes. If you are still with me, congratulations!

## Chapter summary

We started the chapter by dismissing the notion that `1 character == 1 byte`. As the world adopts Unicode (80% of Web sites already use UTF-8), we need to keep the concept of text strings separated from the binary sequences that represent them in files, and Python 3 enforces this separation.

After a brief overview of the binary sequence data types — `bytes`, `bytearray` and `memoryview` — we jumped into encoding and decoding, with a sampling of important codecs, followed by approaches to prevent or deal with the infamous `UnicodeEncodeError`, `UnicodeDecodeError` and the `SyntaxError` caused by wrong encoding in Python source files.

While on the subject of source code, I presented my position on the debate about non-ASCII identifiers: if the maintainers of the code base want to use a human language that has non-ASCII characters, the identifiers should follow suit — unless the code needs to run on Python 2 as well. But if the project aims to attract an international contributor base, identifiers should be made from English words, and then ASCII suffices.

We then considered the theory and practice of encoding detection in the absence of metadata: in theory, it can't be done, but in practice the `Chardet` package pulls it off pretty well for a number of popular encodings. Byte order marks were then presented as the only encoding hint commonly found in UTF-16 and UTF-32 files — sometimes in UTF-8 files as well.

In the next section we demonstrated opening text files, an easy task except for one pitfall: the `encoding=` keyword argument is not mandatory when you open a text file, but it should be. If you fail to specify the encoding, you end up with a program that manages

to generate “plain text” that is incompatible across platforms, due to conflicting default encodings. We then exposed the different encoding settings that Python uses as defaults and how to detect them: `locale.getpreferredencoding()`, `sys.getfilesystemencoding()`, `sys.getdefaultencoding()` and the encodings for the standard I/O files (e.g. `sys.stdout.encoding`). A sad realization for Windows users is that these settings often have distinct values within the same machine, and the values are mutually incompatible; GNU/Linux and OSX users, in contrast, live in a happier place where UTF-8 is the default pretty much everywhere.

Text comparisons are surprisingly complicated because Unicode provides multiple ways of representing some characters, so normalizing is a prerequisite to text matching. In addition to explaining normalization and case folding, we presented some utility functions that you may adapt to your needs, including drastic transformations like removing all accents. We then saw how to sort Unicode text correctly by leveraging the standard `locale` module — with some caveats — and an alternative that does not depend on tricky locale configurations: the external PyUCA package.

Finally we glanced at the Unicode database — a source of metadata about every character — and wrapped up with brief discussion of dual mode APIs — e.g. the `re` and `os` modules — where some functions can be called with `str` or `bytes` arguments, prompting different yet fitting results.

## Further reading

Ned Batchelder’s 2012 PyCon US talk “[Pragmatic Unicode — or — How Do I Stop the Pain?](#)” was outstanding. Ned is so professional that he provides a full transcript of the talk along with the slides and video. Esther Nam and Travis Fischer gave an excellent PyCon 2014 talk “Character encoding and Unicode in Python: How to (ノ ◕◕)ノ (ノ ◕◕)ノ with dignity” ([slides](#), [video](#)) from which I quoted this chapter’s short and sweet epigraph: “Humans use text. Computers speak bytes”. Lennart Regebro — one of this book’s technical reviewers — presents his “Useful Mental Model of Unicode (UMMU)” in the short post [Unconfusing Unicode: What is Unicode?](#). Unicode is a complex standard, so Lennart’s UMMU is a really useful starting point.

The official [Unicode HOWTO](#) in the Python docs approaches the subject from several different angles, from a good historic intro to syntax details, codecs, regular expressions, file names and best practices for Unicode-aware I/O (i.e. the Unicode sandwich), with plenty of additional reference links from each section. [Chapter 4 — Strings](#) of Mark Pilgrim’s awesome book “Dive into Python 3” also provides a very good intro to Unicode support in Python 3. In the same book, [Chapter 15](#) describes how the Chardet library was ported from Python 2 to Python 3, a valuable case study given that the switch from old the `str` to the new `bytes` is the cause of most migration pains, and that is a central concern in a library designed to detect encodings.

If you know Python 2 but are new to Python 3, Guido van Rossum's [What's New In Python 3.0](#) has 15 bullet points that summarize what changed, with lots of links. Guido starts with the blunt statement: "Everything you thought you knew about binary data and Unicode has changed." Armin Ronacher's blog post [The Updated Guide to Unicode on Python](#) is deep and highlights some of the pitfalls of Unicode in Python 3 (Armin is not a big fan of Python 3).

Chapter 2 — Strings and Text — of the Python Cookbook, 3rd. edition (O'Reilly, 2013), by David Beazley and Brian K. Jones, has several recipes dealing with Unicode normalization, sanitizing text, and performing text-oriented operations on byte sequences. Chapter 5 covers files and I/O, and it includes recipe 5.17. — Writing Bytes to a Text File — showing that underlying any text file there is always a binary stream that may be accessed directly when needed. Later in the cookbook the `struct` module is put to use in recipe 6.11 — Reading and Writing Binary Arrays of Structures.

Nick Coghlan's Python Notes blog has two posts very relevant to this chapter: [Python 3 and ASCII Compatible Binary Protocols](#) and [Processing Text Files in Python 3](#). Highly recommended.

Binary sequences are about to gain new constructors and methods in Python 3.5, with one of the current constructor signatures being deprecated (see [PEP 467 — Minor API improvements for binary sequences](#)). Python 3.5 should also see the implementation of [PEP 461 — Adding % formatting to bytes and bytearray](#).

A list of encodings supported by Python is available at [Standard Encodings](#) in the `codecs` module documentation. If you need to get that list programmatically, see how it's done in the [/Tools/unicode/listcodecs.py](#) script that comes with the CPython source code.

Martijn Faassen's [Changing the Python default encoding considered harmful](#) and Tarek Ziade's [sys.setdefaultencoding is evil](#) explain why the default encoding you get from `sys.getdefaultencoding()` should never be changed, even if you discover how.

The books [Unicode Explained](#) by Jukka K. Korpela (O'Reilly, 2006) and [Unicode Demystified](#) by Richard Gillam (Addison-Wesley, 2003) are not Python-specific but were very helpful as I studied Unicode concepts. [Programming with Unicode](#) by Victor Stinner is a free self-published book (Creative Commons BY-SA) covering Unicode in general as well as tools and APIs in the context of the main operating systems and a few programming languages, including Python.

The W3C page [Case Folding: An Introduction and Character Model for the World Wide Web: String Matching and Searching](#) covers normalization concepts, with the former being a gentle introduction and the latter a working draft written in dry standard-speak — the same tone of the [Unicode Standard Annex #15 — Unicode Normalization Forms](#). The [Frequently Asked Questions / Normalization](#) from [Unicode.org](#) is more



readable, as is the [NFC FAQ](#) by Mark Davis — author of several Unicode algorithms and president of the Unicode Consortium at the time of this writing.

## Soapbox


### What is “plain text”?

For anyone who deals non-English text on a daily basis, “plain text” does not imply “ASCII”. The [Unicode Glossary](#) defines *plain text* like this:

Computer-encoded text that consists only of a sequence of code points from a given standard, with no other formatting or structural information.

That definition starts very well, but I don’t agree with the part after the comma. HTML is a great example of a plain text format that carries formatting and structural information. But it’s still plain text because every byte in such a file is there to represent a text character, usually using UTF-8. There are no bytes with non-text meaning, as you can find in a .png or .xls document where most bytes represent packed binary values like RGB values and floating point numbers. In plain text, numbers are represented as sequences of digit characters.

I am writing this book in a plain text format called — ironically — .asciidoc. [AsciiDoc](#) is part of the toolchain of the excellent [Atlas](#) book publishing platform created by O’Reilly Media. AsciiDoc source files are plain text, but they are UTF-8, not ASCII. Otherwise writing this chapter would have been really painful. Despite the name, AsciiDoc is just great.

The world of Unicode is constantly expanding and, at the edges, tool support is not always there. That’s why I had to use images for [Figure 4-1](#), [Figure 4-3](#) and [Figure 4-4](#): not all characters I wanted to show were available in the fonts used to render the book. On the other hand, the Ubuntu 14.04 and OSX 10.9 terminals display them perfectly well — including the Japanese characters for the word “mojibake”: .

### Unicode riddles

Imprecise qualifiers such as “often”, “most” and “usually” seem to pop up whenever I write about Unicode normalization. I regret the lack of more definitive advice, but there are so many exceptions to the rules in Unicode that it is hard to be absolutely positive.

For example, the μ (micro sign) is considered a “compatibility character” but the Ω (ohm) and Å (Ångström) symbols are not. The difference has practical consequences: NFC normalization — recommended for text matching — replaces the Ω (ohm) by Ω (uppercase Greek omega) and the Å (Ångström) by Å (uppercase A with ring above). But as a “compatibility character” the μ (micro sign) is not replaced by the visually identical μ (lowercase Greek mu), except when the stronger NFKC or NFKD normalizations are applied, and these transformations are lossy.

I understand the  $\mu$  (micro sign) is in Unicode because it appears in the `latin1` encoding and replacing it with the Greek mu would break round-trip conversion. After all, that's why the micro sign is a "compatibility character". But if the ohm and Ångström symbols are not in Unicode for compatibility reasons, then why have them at all? There are already code points for the GREEK CAPITAL LETTER OMEGA and the LATIN CAPITAL LETTER A WITH RING ABOVE which look the same and replace them on NFC normalization. Go figure.

My take after many hours studying Unicode: it is hugely complex and full of special cases, reflecting the wonderful variety of human languages and the politics of industry standards.

### How are `str` represented in RAM?

The official Python docs avoid the issue of how the code points of a `str` are stored in memory. This is, after all, an implementation detail. In theory it doesn't matter: whatever the internal representation, every `str` must be encoded to bytes on output.

In memory, Python 3 stores each `str` as a sequence of code points using a fixed number of bytes per code point, to allow efficient direct access to any character or slice.

Before Python 3.3, CPython could be compiled to use either 16 or 32 bits per code point in RAM; the former was a "narrow build", the latter a "wide build". To know which you have, check the value of `sys.maxunicode`: 65535 implies a "narrow build" that can't handle code points above U+FFFF transparently. A "wide build" doesn't have this limitation, but consumes a lot of memory: 4 bytes per character, even while the vast majority of code points for Chinese ideographs fit in 2 bytes. Neither option was great, so you had to choose depending on your needs.

Since Python 3.3, when creating a new `str` object, the interpreter checks the characters in it and chooses the most economic memory layout that is suitable for that particular `str`: if there are only characters in the `latin1` range, that `str` will use just one byte per code point. Otherwise 2 or 4 bytes per code point may be used, depending on the `str`. This is a simplification, for the full details look up [PEP 393 — Flexible String Representation](#).

The flexible string representation is similar to the way the `int` type works in Python 3: if the integer fits in a machine word, it is stored in one machine word. Otherwise, the interpreter switches to a variable-length representation like that of the Python 2 `long` type. It is nice to see the spread of good ideas.