
Unicode Text Versus Bytes

Humans use text. Computers speak bytes.

—Esther Nam and Travis Fischer, “Character Encoding and Unicode in Python”¹

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 the kind of work you do with Python, you may think that understanding Unicode is not important. That’s unlikely, but anyway 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 did 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`
- Encodings 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`

What's New in This Chapter

Support for Unicode in Python 3 has been comprehensive and stable, so the most notable addition is “[Finding Characters by Name](#)” on [page 151](#), describing a utility for searching the Unicode database—a great way to find circled digits and smiling cats from the command line.

One minor change worth mentioning is the Unicode support on Windows, which is better and simpler since Python 3.6, as we’ll see in “[Beware of Encoding Defaults](#)” on [page 134](#).

Let’s start with the not-so-new, but fundamental concepts of characters, code points, and bytes.



For the second edition, I expanded the section about the `struct` module and published it online at “[Parsing binary records with struct](#)”, in the *[fluentpython.com](#)* companion website.

There you will also find “[Building Multi-character Emojis](#)”, describing how to make country flags, rainbow flags, people with different skin tones, and diverse family icons by combining Unicode characters.

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 2021, the best definition of “character” we have is a Unicode character. Accordingly, the items we 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 we got 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 hex digits with a “U+” prefix, from U+0000 to U+10FFFF. 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 13% of the valid code points have characters assigned to them in Unicode 13.0.0, the standard used in Python 3.10.0b4.
- 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 the letter 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, UTF-8 requires three bytes—`\xe2\x82\xac`—to encode the Euro sign (U+20AC), but in UTF-16LE the same code point is encoded as two bytes: `\xac\x20`.

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

Example 4-1. Encoding and decoding

```
>>> s = 'café'
>>> len(s) ❶
4
>>> b = s.encode('utf8') ❷
>>> b
b'caf\xc3\xa9' ❸
>>> len(b) ❹
5
>>> b.decode('utf8') ❺
'café'
```

- ❶ The str 'café' has four Unicode characters.
- ❷ Encode str to bytes using UTF-8 encoding.
- ❸ bytes literals have 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 help 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 way back in Python 2.6.² The Python documentation sometimes uses the generic term “byte string” to refer to both `bytes` and `bytearray`. I avoid that confusing term.

Each item in `bytes` or `bytearray` is an integer from 0 to 255, and not a one-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
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)`.

² Python 2.6 and 2.7 also had `bytes`, but it was just an alias to the `str` type.

- ③ Slices of bytes are also bytes—even slices of a single byte.
- ④ There is no literal syntax for bytearray: they are shown as bytearray() with a bytes literal as argument.
- ⑤ A slice of bytearray is also a bytearray.



The fact that `my_bytes[0]` retrieves an `int` but `my_bytes[:1]` returns a bytes sequence of length 1 is only surprising because we are used to Python's `str` type, where `s[0] == s[:1]`. For all other sequence types in Python, 1 item is not the same as a slice of length 1.

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

- For bytes with decimal codes 32 to 126—from space to ~ (tilde)—the ASCII character itself is used.
- For bytes corresponding to tab, newline, carriage return, and \, the escape sequences `\t`, `\n`, `\r`, and `\\` are used.
- If both string delimiters ' and " appear in the byte sequence, the whole sequence is delimited by ', and any ' inside are escaped as \'.³
- For other byte values, a 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 those that depend on Unicode data, including `case fold`, `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

³ Trivia: the ASCII “single quote” character that Python uses by default as the string delimiter is actually named APOSTROPHE in the Unicode standard. The real single quotes are asymmetric: left is U+2018 and right is U+2019.

on binary sequences, if the regex is compiled from a binary sequence instead of a `str`. Since Python 3.5, the `%` operator works with binary sequences again.⁴

Binary sequences have a class method that `str` doesn't have, called `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
- An object that implements the buffer protocol (e.g., `bytes`, `bytearray`, `memory view`, `array.array`) that copies the bytes from the source object to the newly created binary sequence



Until Python 3.5, it was also possible to call `bytes` or `bytearray` with a single integer to create a binary sequence of that size initialized with null bytes. This signature was deprecated in Python 3.5 and removed in Python 3.6. See [PEP 467—Minor API improvements for binary sequences](#).

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 5 short integers.

⁴ It did not work in Python 3.0 to 3.4, causing much pain to developers dealing with binary data. The reversal is documented in [PEP 461—Adding % formatting to bytes and bytearray](#).

Creating a bytes or bytearray object from any buffer-like source will always copy the bytes. In contrast, memoryview objects let you share memory between binary data structures, as we saw in “Memory Views” on page 62.

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

Basic Encoders/Decoders

The Python distribution bundles more than 100 *codecs* (encoder/decoders) 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 4-4 shows the same text encoded as three different byte sequences.

Example 4-4. 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\xf1o'
utf_8   b'El Ni\xc3\xb1o'
utf_16  b'\xff\xfe\x00l\x00 \x00N\x00i\x00\xf1\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, multibyte 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 7 different encodings (asterisks indicate that the character cannot be represented in that encoding).

All those asterisks in [Figure 4-1](#) make clear that some encodings, like ASCII and even the multibyte 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 useful `latin1` superset created by Microsoft, adding useful symbols like curly quotes and € (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 multibyte encodings for Asian languages.

`utf-8`

The most common 8-bit encoding on the web, by far, as of July 2021, “[W³Techs: Usage statistics of character encodings for websites](#)” claims that 97% of sites use UTF-8, up from 81.4% when I wrote this paragraph in the first edition of this book in September 2014.

`utf-16le`

One form of the UTF 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 used in many systems despite being deprecated since the last century because it only supports code points up to U+FFFF. As of 2021, more than 57% of the allocated code points are above U+FFFF, including the all-important emojis.

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

Understanding Encode/Decode Problems

Although there is a generic `UnicodeError` exception, the error reported by Python is usually more specific: either a `UnicodeEncodeError` (when converting `str` to binary sequences) or a `UnicodeDecodeError` (when reading binary sequences into `str`). Loading Python modules may also raise `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 a `UnicodeEncodeError`, a `UnicodeDecodeError`, or some other error (e.g., `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 4-5](#).

Example 4-5. 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'` string.
- ❸ `cp437` can't encode the `'ã'` (“a” with tilde). The default error handler — `'strict'` — raises `UnicodeEncodeError`.
- ❹ The `error='ignore'` handler skips characters that cannot be encoded; this is usually a very bad idea, leading to silent data loss.
- ❺ When encoding, `error='replace'` substitutes unencodable characters with `'?'`; data is also lost, but users will get a clue that something is amiss.
- ❻ `'xmlcharrefreplace'` replaces unencodable characters with an XML entity. If you can't use UTF, and you can't afford to lose data, this is the only option.



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](#).

ASCII is a common subset to all the encodings that I know about, therefore encoding should always work if the text is made exclusively of ASCII characters. Python 3.7 added a new boolean method `str.isascii()` to check whether your Unicode text is 100% pure ASCII. If it is, you should be able to encode it to bytes in any encoding without raising `UnicodeEncodeError`.

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'`, and `'koi8_r'` are able to decode any stream of bytes, including random noise, without reporting errors. Therefore, if your program assumes the wrong 8-bit encoding, it will silently decode garbage.



Garbled characters are known as gremlins or mojibake (文字化け —Japanese for “transformed text”).

Example 4-6 illustrates how using the wrong codec may produce gremlins or a `UnicodeDecodeError`.

Example 4-6. Decoding from `str` to bytes: success and error handling

```
>>> octets = b'Montr\xe9al' ❶
>>> octets.decode('cp1252') ❷
'Montréal'
>>> octets.decode('iso8859_7') ❸
'Montrial'
>>> octets.decode('koi8_r') ❹
'MontrMal'
>>> octets.decode('utf_8') ❺
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') ❻
'Montr  al'
```

- ❶ The word “Montréal” encoded as `latin1`; `'\xe9'` is the byte for “é”.
- ❷ Decoding with Windows 1252 works because it is a 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 “И”.
- ❺ The `'utf_8'` codec detects that `octets` is not valid UTF-8, and raises `UnicodeDecodeError`.
- 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. 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 https://python.org/dev/peps/pep-0263/
for details
```

Because UTF-8 is widely deployed in GNU/Linux and macOS 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 source is UTF-8 across all platforms.

To fix this problem, add a magic coding comment at the top of the file, as shown in [Example 4-7](#).

Example 4-7. ola.py: “Hello, World!” in Portuguese

```
# coding: cp1252

print('Olá, 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.

Suppose you have a text file, be it source code or poetry, but you don’t know its encoding. How do you detect the actual encoding? Answers in the next section.

How to Discover the Encoding of a Byte Sequence

How do you find 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.

Leo's Hack for Guessing UTF-8 Decoding

(The next paragraphs come from a note left by tech reviewer Leonardo Rochael in the draft of this book.)

The way UTF-8 was designed, it's almost impossible for a random sequence of bytes, or even a nonrandom sequence of bytes coming from a non-UTF-8 encoding, to be decoded accidentally as garbage in UTF-8, instead of raising `UnicodeDecodeError`.

The reasons for this are that UTF-8 escape sequences never use ASCII characters, and these escape sequences have bit patterns that make it very hard for random data to be valid UTF-8 by accident.

So if you can decode some bytes containing codes > 127 as UTF-8, it's probably UTF-8.

In dealing with Brazilian online services, some of which were attached to legacy back-ends, I've had, on occasion, to implement a decoding strategy of trying to decode via UTF-8 and treat a `UnicodeDecodeError` by decoding via `cp1252`. It was ugly but effective.

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 more 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 guess one of more than 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 4-4**, you may have noticed a couple of extra bytes at the beginning of a UTF-16 encoded sequence. Here they are again:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\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 invisible character ZERO WIDTH NO-BREAK SPACE (U+FEFF). On a little-endian system, that is encoded as `b'\xff\xfe'` (decimal 255, 254). Because, by design, there is no U+FFFE character in Unicode, 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 Unicode 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 a UTF-8 file, otherwise it assumes the content is encoded with a Windows code page. This UTF-8 encoding with BOM is called UTF-8-SIG in Python’s codec registry. The character U+FEFF

encoded in UTF-8-SIG 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.



Caleb's Tip about UTF-8-SIG

Caleb Hattingh—one of the tech reviewers—suggests always using the UTF-8-SIG codec when reading UTF-8 files. This is harmless because UTF-8-SIG reads files with or without a BOM correctly, and does not return the BOM itself. When writing, I recommend using UTF-8 for general interoperability. For example, Python scripts can be made executable in Unix systems if they start with the comment: `#!/usr/bin/env python3`. The first two bytes of the file must be `b'#!'` for that to work, but the BOM breaks that convention. If you have a specific requirement to export data to apps that need the BOM, use UTF-8-SIG but be aware that Python's [codecs documentation](#) says: “In UTF-8, the use of the BOM is discouraged and should generally be avoided.”

We now move on to handling text files in Python 3.

Handling Text Files

The best practice for handling text I/O is the “Unicode sandwich” ([Figure 4-2](#)).⁵ This means that bytes should be decoded to `str` as early as possible on input (e.g., when opening a file for reading). The “filling” 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.

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

⁵ I first saw the term “Unicode sandwich” in Ned Batchelder’s excellent “[Pragmatic Unicode](#)” talk at US PyCon 2012.

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.

Consider the console session in [Example 4-8](#). Can you spot the bug?

Example 4-8. 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©'
```

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

I ran [Example 4-8](#) on Python 3.8.1, 64 bits, on Windows 10 (build 18363). The same statements running on recent GNU/Linux or macOS 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 4-8](#) is that the write function in the first statement reports that four characters were written, but in the next line five characters are read. [Example 4-9](#) is an extended version of [Example 4-8](#), explaining that and other details.

Example 4-9. Closer inspection of [Example 4-8](#) running on Windows reveals the bug and how to fix it

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café') ❷
4
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size ❸
5
>>> fp2 = open('cafe.txt')
>>> fp2 ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ❺
'cp1252'
>>> fp2.read() ❻
'cafÃ©'
>>> fp3 = open('cafe.txt', encoding='utf_8') ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read() ❽
'café'
>>> fp4 = open('cafe.txt', 'rb') ❾
>>> fp4
<_io.BufferedReader name='cafe.txt'>
>>> fp4.read() ❿
b'caf\xc3\xa9'
```

- ❶ By default, open uses text mode and returns a TextIOWrapper object with a specific encoding.
- ❷ The write method on a TextIOWrapper returns the number of Unicode characters written.
- ❸ os.stat says the file has 5 bytes; UTF-8 encodes 'é' as 2 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.

- ❹ In the Windows cp1252 encoding, the byte 0xc3 is an “Ã” (A with tilde), and 0xa9 is the copyright sign.
- ❺ Opening the same file with the correct encoding.
- ❻ The expected result: the same four Unicode characters for 'café'.
- ❼ The 'rb' flag opens a file for reading in binary mode.
- ❽ The returned object is a `BufferedReader` and not a `TextIOWrapper`.
- ❾ 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 128). Ordinary code should only use binary mode to open binary files, like raster images.

The problem in [Example 4-9](#) 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.

Beware of Encoding Defaults

Several settings affect the encoding defaults for I/O in Python. See the `default_encodings.py` script in [Example 4-10](#).

Example 4-10. Exploring encoding defaults

```
import locale
import sys

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(f'{expression:>30} -> {value!r}')

```

The output of [Example 4-10](#) on GNU/Linux (Ubuntu 14.04 to 19.10) and macOS (10.9 to 10.14) 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 4-11](#).

Example 4-11. Default encodings on Windows 10 PowerShell (output is the same on cmd.exe)

```

> chcp ❶
Active code page: 437
> python default_encodings.py ❷
locale.getpreferredencoding() -> 'cp1252' ❸
type(my_file) -> <class '_io.TextIOWrapper'>
my_file.encoding -> 'cp1252' ❹
sys.stdout.isatty() -> True ❺
sys.stdout.encoding -> 'utf-8' ❻
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'

```

- ❶ chcp shows the active code page for the console: 437.
- 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.

- 5 The output is going to the console, so `sys.stdout.isatty()` is `True`.
- 6 Now, `sys.stdout.encoding` is not the same as the console code page reported by `chcp`!

Unicode support in Windows itself, and in Python for Windows, got better since I wrote the first edition of this book. [Example 4-11](#) used to report four different encodings in Python 3.4 on Windows 7. The encodings for `stdout`, `stdin`, and `stderr` used to be the same as the active code page reported by the `chcp` command, but now they're all `utf-8` thanks to [PEP 528—Change Windows console encoding to UTF-8](#) implemented in Python 3.6, and Unicode support in PowerShell in *cmd.exe* (since Windows 1809 from October 2018).⁶ It's weird that `chcp` and `sys.stdout.encoding` say different things when `stdout` is writing to the console, but it's great that now we can print Unicode strings without encoding errors on Windows—unless the user redirects output to a file, as we'll soon see. That does not mean all your favorite emojis will appear in the console: that also depends on the font the console is using.

Another change was [PEP 529—Change Windows filesystem encoding to UTF-8](#), also implemented in Python 3.6, which changed the filesystem encoding (used to represent names of directories and files) from Microsoft's proprietary MBCS to UTF-8.

However, if the output of [Example 4-10](#) 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—but `sys.stdin.encoding` and `sys.stderr.encoding` remain `utf-8`.



In [Example 4-12](#) I use the `'\N{'` escape for Unicode literals, where we write the official name of the character inside the `\N{'`. It's rather verbose, but explicit and safe: Python raises `SyntaxError` if the name doesn't exist—much better than writing a hex number that could be wrong, but you'll only find out much later. You'd probably want to write a comment explaining the character codes anyway, so the verbosity of `\N{'` is easy to accept.

This means that a script like [Example 4-12](#) works when printing to the console, but may break when output is redirected to a file.

⁶ Source: "[Windows Command-Line: Unicode and UTF-8 Output Text Buffer](#)".

Example 4-12. stdout_check.py

```
import sys
from unicodedata import name

print(sys.version)
print()
print('sys.stdout.isatty():', sys.stdout.isatty())
print('sys.stdout.encoding:', sys.stdout.encoding)
print()

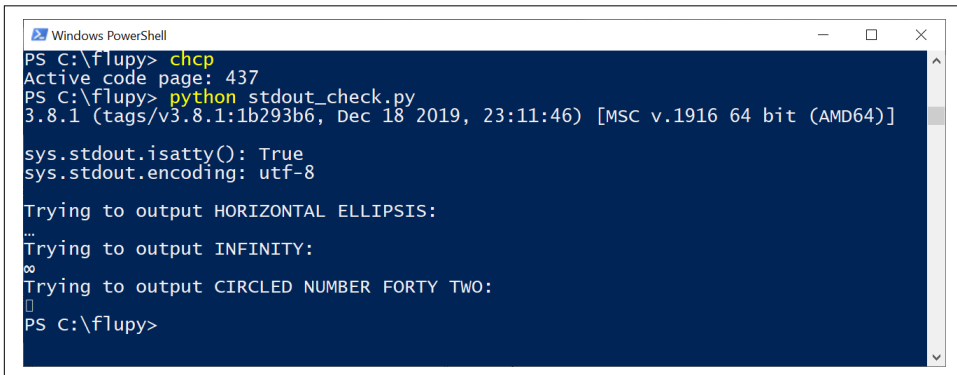
test_chars = [
    '\N{HORIZONTAL ELLIPSIS}',      # exists in cp1252, not in cp437
    '\N{INFINITY}',                  # exists in cp437, not in cp1252
    '\N{CIRCLED NUMBER FORTY TWO}', # not in cp437 or in cp1252
]

for char in test_chars:
    print(f'Trying to output {name(char)}:')
    print(char)
```

Example 4-12 displays the result of `sys.stdout.isatty()`, the value of `sys.stdout.encoding`, and these three characters:

- '...' HORIZONTAL ELLIPSIS—exists in CP 1252 but not in CP 437.
- '∞' INFINITY—exists in CP 437 but not in CP 1252.
- 'Ⓣ' CIRCLED NUMBER FORTY TWO—doesn't exist in CP 1252 or CP 437.

When I run `stdout_check.py` on PowerShell or `cmd.exe`, it works as captured in Figure 4-3.



```
Windows PowerShell
PS C:\flupy> chcp
Active code page: 437
PS C:\flupy> python stdout_check.py
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

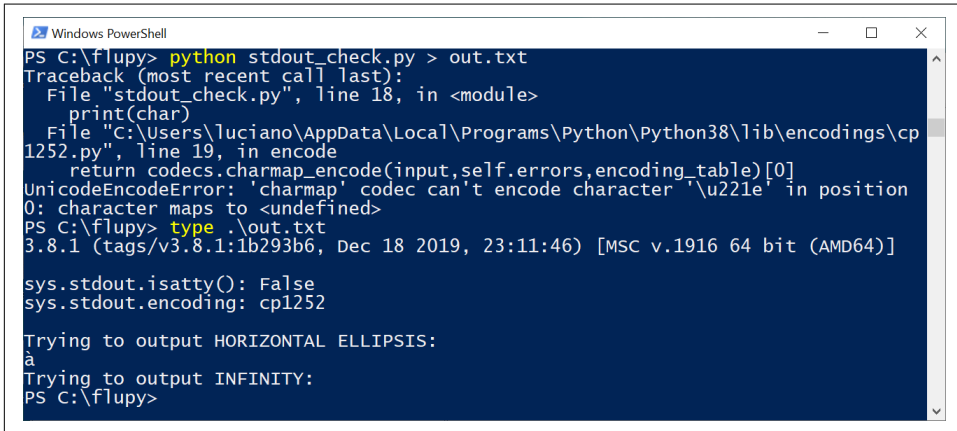
sys.stdout.isatty(): True
sys.stdout.encoding: utf-8

Trying to output HORIZONTAL ELLIPSIS:
...
Trying to output INFINITY:
∞
Trying to output CIRCLED NUMBER FORTY TWO:
Ⓣ
PS C:\flupy>
```

Figure 4-3. Running stdout_check.py on PowerShell.

Despite `chcp` reporting the active code as 437, `sys.stdout.encoding` is UTF-8, so the HORIZONTAL ELLIPSIS and INFINITY both output correctly. The CIRCLED NUMBER FORTY TWO is replaced by a rectangle, but no error is raised. Presumably it is recognized as a valid character, but the console font doesn't have the glyph to display it.

However, when I redirect the output of `stdout_check.py` to a file, I get [Figure 4-4](#).



```
PS C:\flupy> python stdout_check.py > out.txt
Traceback (most recent call last):
  File "stdout_check.py", line 18, in <module>
    print(char)
  File "C:\Users\luciano\AppData\Local\Programs\Python\Python38\lib\encodings\cp
1252.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u221e' in position
0: character maps to <undefined>
PS C:\flupy> type .\out.txt
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

sys.stdout.isatty(): False
sys.stdout.encoding: cp1252

Trying to output HORIZONTAL ELLIPSIS:
à
Trying to output INFINITY:
PS C:\flupy>
```

Figure 4-4. Running `stdout_check.py` on PowerShell, redirecting output.

The first problem demonstrated by [Figure 4-4](#) is the `UnicodeEncodeError` mentioning character `'\u221e'`, because `sys.stdout.encoding` is `'cp1252'`—a code page that doesn't have the INFINITY character.

Reading `out.txt` with the `type` command—or a Windows editor like VS Code or Sublime Text—shows that instead of HORIZONTAL ELLIPSIS, I got `'à'` (LATIN SMALL LETTER A WITH GRAVE). As it turns out, the byte value `0x85` in CP 1252 means `'...'`, but in CP 437 the same byte value represents `'à'`. So it seems the active code page does matter, not in a sensible or useful way, but as partial explanation of a bad Unicode experience.



I used a laptop configured for the US market, running Windows 10 OEM to run these experiments. Windows versions localized for other countries may have different encoding configurations. For example, in Brazil the Windows console uses code page 850 by default—not 437.

To wrap up this maddening issue of default encodings, let's give a final look at the different encodings in [Example 4-11](#):

- If you omit the encoding argument when opening a file, the default is given by `locale.getpreferredencoding()` ('cp1252' in [Example 4-11](#)).
- The encoding of `sys.stdout|stdin|stderr` used to be set by the `PYTHONIOENCODING` environment variable before Python 3.6—now that variable is ignored, unless `PYTHONLEGACYWINDOWSSTDIO` is set to a nonempty string. Otherwise, the encoding for standard I/O is UTF-8 for interactive I/O, or defined by `locale.getpreferredencoding()` if the output/input is redirected to/from a file.
- `sys.getdefaultencoding()` is used internally by Python in implicit conversions of binary data to/from `str`. Changing this setting is not supported.
- `sys.getfilesystemencoding()` is used to encode/decode filenames (not file contents). It is used when `open()` gets a `str` argument for the filename; if the filename is given as a `bytes` argument, it is passed unchanged to the OS API.



On GNU/Linux and macOS, 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 code pages 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.

You will avoid a lot of pain if you follow the advice of the Unicode sandwich and always are explicit about the encodings in your programs. 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 Reliable 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 four or five 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
```

Placing COMBINING ACUTE ACCENT (U+0301) after “e” renders “é”. 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 `unicodedata.normalize()`. The first argument to that function is one of four strings: 'NFC', 'NFD', 'NFKC', and 'NFKD'. Let's start with the first two.

Normalization Form C (NFC) 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, as the next example shows:

```
>>> from unicodedata import normalize
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> 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
```

Keyboard drivers usually generate composed characters, so text typed by users will be in NFC by default. However, to be safe, it may be good to normalize 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 (Ω) unit of electrical resistance is normalized to the Greek upper-case omega. They are visually identical, but they compare as 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 other two normalization forms are NFKC and NFKD, where the letter K 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, which includes it, even though the same character is part of the Greek alphabet with code point 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).⁷

Here is how the NFKC works in practice:

```
>>> from unicodedata import normalize, name
>>> half = '\N{VULGAR FRACTION ONE HALF}'
>>> print(half)
½
>>> normalize('NFKC', half)
'1/2'
```

⁷ 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 Greek characters.

```

>>> for char in normalize('NFKC', half):
...     print(char, name(char), sep='\t')
...
1 DIGIT ONE
/ FRACTION SLASH
2 DIGIT TWO
>>> 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. An application could store '4²' as '4²', but the `normalize` function knows nothing about formatting. Therefore, NFKC or NFKD may lose or distort information, but they can produce convenient intermediate representations for searching and indexing.

Unfortunately, with Unicode everything is always more complicated than it first seems. For the VULGAR FRACTION ONE HALF, the NFKC normalization produced 1 and 2 joined by FRACTION SLASH, instead of SOLIDUS, a.k.a. “slash”—the familiar character with ASCII code decimal 47. Therefore, searching for the three-character ASCII sequence '1/2' would not find the normalized Unicode sequence.



NFKC and NFKD normalization cause data loss and should be applied only in special cases like search and indexing, and not for permanent storage of text.

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.

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 lowercase 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')
```

There are nearly 300 code points for which `str.casefold()` and `str.lower()` return different results.

As usual with anything related to Unicode, case folding is a hard 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\u00301'
>>> 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—which are 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 Wikipedia article about the city of São Paulo:

```
https://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 easier to recognize, even if it is not the right spelling:

`https://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. simplify.py: function to remove all combining marks

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

- ❶ Only the letters “è”, “ç”, and “í” were replaced.
- ❷ Both “ξ” and “é” 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 `simplify.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
    preserve = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base: ❷
            continue # ignore diacritic on Latin base char
        preserve.append(c) ❸
        # if it isn't a combining char, it's a new base char
        if not unicodedata.combining(c): ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(preserve)
    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 (e.g., curly quotes, em dashes, bullets, etc.) into ASCII equivalents. This is what the function `asciiize` does in [Example 4-17](#).

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

```
single_map = str.maketrans("''',^<€'“”•—~}", ❶
                           "''''f"'^<'!'"---~>""")

multi_map = str.maketrans({ ❷
    '€': 'EUR',
    '…': '...',
    'Æ': 'AE',
    'æ': 'ae',
    'Œ': 'OE',
    'œ': 'oe',
```

```

    '™': '(TM)',
    '‰': '<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 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 ☿tker™ caffè latte • bowl of açai."
>>> dewinize(order)
'Herr Voß: - ½ cup of 0Etker(TM) caffè latte - bowl of açai.' ❶
>>> asciize(order)
'Herr Voss: - 1/2 cup of 0Etker(TM) caffè latte - bowl of acai.' ❷

```

- ❶ `dewinize` replaces curly quotes, bullets, and [™] (trademark 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 *simplify.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.

Now let's sort out Unicode 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.⁸ 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.”

⁸ 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. The sequence of commands in [Example 4-19](#) may work for you.

Example 4-19. `locale_sort.py`: using the `locale.strxfrm` function as the sort key

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

Running [Example 4-19](#) on GNU/Linux (Ubuntu 19.10) with the `pt_BR.UTF-8` locale installed, I get the correct result:

```
'pt_BR.UTF-8'
['açai', '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 some 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 afterward.
- 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.
- The locale must be correctly implemented by the makers of the OS. I was successful on Ubuntu 19.10, but not on macOS 10.14. On macOS, 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 incorrect result as `sorted(fruits)` did. I also tried the `fr_FR`, `es_ES`, and `de_DE` locales on macOS, but `locale.strxfrm` never did its job.⁹

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.

⁹ 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 Macintosh with macOS 10.9. In summary: your mileage may vary.

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 the Unicode Collation Algorithm (UCA). **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 simple and works on GNU/Linux, macOS, and Windows, at least with my small sample.

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.org***.



PyICU: Miro's Recommendation for Unicode Sorting

(Tech reviewer Miroslav Šedivý is a polyglot and an expert on Unicode. This is what he wrote about *pyuca*.)

pyuca has one sorting algorithm that does not respect the sorting order in individual languages. For instance, Ä in German is between A and B, while in Swedish it comes after Z. Have a look at **PyICU** that works like locale without changing the locale of the process. It is also needed if you want to change the case of iİ/iI in Turkish. PyICU includes an extension that must be compiled, so it may be harder to install in some systems than *pyuca*, which is just Python.

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

The Unicode Database

The Unicode standard provides an entire database—in the form of several structured text files—that includes not only the table mapping code points to character names,

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



The `unicodedata.category(char)` function returns the two-letter category of `char` from the Unicode database. The higher-level `str` methods are easier to use. For example, `label.isalpha()` returns `True` if every character in `label` belongs to one of these categories: `Lm`, `Lt`, `Lu`, `Ll`, or `Lo`. To learn what those codes mean, see “[General Category](#)” in the English Wikipedia’s “[Unicode character property](#)” article.

Finding Characters by Name

The `unicodedata` module has functions to retrieve character metadata, including `unicodedata.name()`, which returns a character’s official name in the standard. [Figure 4-5](#) demonstrates that function.¹⁰

```
>>> from unicodedata import name
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> name('ã')
'LATIN SMALL LETTER A WITH TILDE'
>>> name('♚')
'BLACK CHESS QUEEN'
>>> name('😺')
'GRINNING CAT FACE WITH SMILING EYES'
```

Figure 4-5. Exploring `unicodedata.name()` in the Python console.

You can use the `name()` function to build apps that let users search for characters by name. [Figure 4-6](#) demonstrates the `cf.py` command-line script that takes one or more words as arguments, and lists the characters that have those words in their official Unicode names. The full source code for `cf.py` is in [Example 4-21](#).

¹⁰ That’s an image—not a code listing—because emojis are not well supported by O’Reilly’s digital publishing toolchain as I write this.

```
$ ./cf.py cat smiling
U+1F638 🐱 GRINNING CAT FACE WITH SMILING EYES
U+1F63A 🐱 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 🐱 SMILING CAT FACE WITH HEART-SHAPED EYES
```

Figure 4-6. Using *cf.py* to find smiling cats.



Emoji support varies widely across operating systems and apps. In recent years the macOS terminal offers the best support for emojis, followed by modern GNU/Linux graphic terminals. Windows *cmd.exe* and PowerShell now support Unicode output, but as I write this section in January 2020, they still don't display emojis—at least not “out of the box.” Tech reviewer Leonardo Rochaël told me about a new, open source [Windows Terminal by Microsoft](#), which may have better Unicode support than the older Microsoft consoles. I did not have time to try it.

In [Example 4-21](#), note the `if` statement in the `find` function using the `.issubset()` method to quickly test whether all the words in the query set appear in the list of words built from the character's name. Thanks to Python's rich set API, we don't need a nested `for` loop and another `if` to implement this check.

Example 4-21. *cf.py*: the character finder utility

```
#!/usr/bin/env python3
import sys
import unicodedata

START, END = ord(' '), sys.maxunicode + 1 ❶

def find(*query_words, start=START, end=END): ❷
    query = {w.upper() for w in query_words} ❸
    for code in range(start, end):
        char = chr(code) ❹
        name = unicodedata.name(char, None) ❺
        if name and query.issubset(name.split()): ❻
            print(f'U+{code:04X}\t{char}\t{name}') ❼

def main(words):
    if words:
        find(*words)
    else:
        print('Please provide words to find.')

if __name__ == '__main__':
    main(sys.argv[1:])
```

- ❶ Set defaults for the range of code points to search.
- ❷ `find` accepts `query_words` and optional keyword-only arguments to limit the range of the search, to facilitate testing.
- ❸ Convert `query_words` into a set of uppercased strings.
- ❹ Get the Unicode character for code.
- ❺ Get the name of the character, or `None` if the code point is unassigned.
- ❻ If there is a name, split it into a list of words, then check that the query set is a subset of that list.
- ❼ Print out line with code point in U+9999 format, the character, and its name.

The `unicodedata` module has other interesting functions. Next, we'll see a few that are related to getting information from characters that have numeric meaning.

Numeric Meaning of Characters

The `unicodedata` module includes functions to check whether a Unicode character represents a number and, if so, its numeric value for humans—as opposed to its code point number. [Example 4-22](#) shows the use of `unicodedata.name()` and `unicodedata.numeric()`, along with the `.isdecimal()` and `.isnumeric()` methods of `str`.

Example 4-22. 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(f'U+{ord(char):04x}',          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
          f'{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 4-22](#) gives you [Figure 4-7](#), if your terminal font has all those glyphs.

```

$ python3 numerics_demo.py
U+0031  1      re_dig isdig isnum  1.00  DIGIT ONE
U+00bc  ¼      -      isdig 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     -      isdig isnum 12.00  ROMAN NUMERAL TWELVE
U+2466  ⑦      -      isdig isnum  7.00  CIRCLED DIGIT SEVEN
U+2480  ⑬      -      isdig isnum 13.00  PARENTHESIZED NUMBER THIRTEEN
U+3285  ⑆      -      isdig isnum  6.00  CIRCLED IDEOGRAPH SIX
$

```

Figure 4-7. macOS terminal showing numeric characters and metadata about them; `re_dig` means the character matches the regular expression `r'\d'`.

The sixth column of [Figure 4-7](#) 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 4-7](#) shows that the regular expression `r'\d'` matches the digit “1” and the Devanagari digit 3, 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 on PyPI was designed to eventually replace `re` and provides better Unicode support.¹¹ We’ll come back to the `re` module in the next section.

¹¹ Although it was not better than `re` at identifying digits in this particular sample.

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 [unicode data module](#).

Next we'll take a quick look at dual-mode APIs offering functions that accept `str` or `bytes` arguments with special handling depending on the type.

Dual-Mode `str` and `bytes` APIs

Python's standard library has functions that accept `str` or `bytes` arguments and behave differently depending on the type. Some examples can be found in the `re` and `os` modules.

`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 4-23](#) and [Figure 4-8](#) compare how letters, ASCII digits, superscripts, and Tamil digits are matched by `str` and `bytes` patterns.

Example 4-23. `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(f'Text\n {text_str!r}')
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.

- ③ Unicode text to search, containing the Tamil digits for 1729 (the logical line continues until the right parenthesis token).
- ④ This string is joined to the previous one at compile time (see “2.4.2. String literal concatenation” in *The Python Language Reference*).
- ⑤ A bytes string is needed to search with the bytes regular expressions.
- ⑥ The `str` pattern `r'\d+'` matches the Tamil and ASCII digits.
- ⑦ The bytes pattern `rb'\d+'` matches only the ASCII bytes for digits.
- ⑧ The `str` pattern `r'\w+'` matches the letters, superscripts, Tamil, and ASCII digits.
- ⑨ The bytes pattern `rb'\w+'` matches only the ASCII bytes for letters and digits.

```

1. bash
$ python3 ramanujan.py
Text
'Ramanujan saw க௭௨௯ as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
str : ['க௭௨௯', '1729', '1', '12', '9', '10']
bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
str : ['Ramanujan', 'saw', 'க௭௨௯', 'as', '1729', '1³', '12³', '9³', '10³']
bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$

```

Figure 4-8. Screenshot of running `ramanujan.py` from *Example 4-23*.

Example 4-23 is a trivial example to make one point: you can use regular expressions on `str` and `bytes`, but in the second case, bytes outside the ASCII range are treated as nondigits and nonword 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 in os Functions

The GNU/Linux kernel is not Unicode savvy, so in the real world you may find filenames 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.

In order to work around this issue, all `os` module functions that accept filenames or pathnames 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) filenames 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 pathname, no matter how many gremlins you may find. See [Example 4-24](#).

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

```
>>> os.listdir('.') ❶
['abc.txt', 'digits-of-π.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 filenames or pathnames, the `os` module provides special encoding and decoding functions `os.fsencode(name_or_path)` and `os.fsdecode(name_or_path)`. Both of these functions accept an argument of type `str`, `bytes`, or an object implementing the `os.PathLike` interface since Python 3.6.

Unicode is a deep rabbit hole. Time to wrap up our exploration of `str` and `bytes`.

Chapter Summary

We started the chapter by dismissing the notion that `1 character == 1 byte`. As the world adopts Unicode, 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.

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. 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 macOS users, in contrast, live in a happier place where UTF-8 is the default pretty much everywhere.

Unicode provides multiple ways of representing some characters, so normalizing is a prerequisite for 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.

We leveraged the Unicode database to program a command-line utility to search for characters by name—in 28 lines of code, thanks to the power of Python. We glanced at other Unicode metadata, and had a brief overview of dual-mode APIs where some functions can be called with `str` or `bytes` arguments, producing different 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.

“Character encoding and Unicode in Python: How to (ノ ◕□◕)ノ へ ￣￣ with dignity” ([slides](#), [video](#)) was the excellent PyCon 2014 talk by Esther Nam and Travis Fischer, where I found this chapter's pithy epigraph: “Humans use text. Computers speak bytes.”

Lennart Regebro—one of the technical reviewers for the first edition of this book—shares 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, filenames, 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](#)* (Apress) 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 the old `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 ed.](#)* (O’Reilly), 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.

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.

The books *[Unicode Explained](#)* by Jukka K. Korpela (O’Reilly) and *[Unicode Demystified](#)* by Richard Gillam (Addison-Wesley) 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 pages “[Case Folding: An Introduction](#)” and “[Character Model for the World Wide Web: String Matching](#)” cover normalization concepts, with the former being a gentle introduction and the latter a working group note written in dry

standard-speak—the same tone of the “Unicode Standard Annex #15—Unicode Normalization Forms”. The “Frequently Asked Questions, Normalization” section 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.

In 2016, the Museum of Modern Art (MoMA) in New York added to its collection the original emoji, the 176 emojis designed by Shigetaka Kurita in 1999 for NTT DOCOMO—the Japanese mobile carrier. Going further back in history, *Emojipedia* published “Correcting the Record on the First Emoji Set”, crediting Japan’s SoftBank for the earliest known emoji set, deployed in cell phones in 1997. SoftBank’s set is the source of 90 emojis now in Unicode, including U+1F4A9 (PILE OF POO). Matthew Rothenberg’s *emojitracker.com* is a live dashboard showing counts of emoji usage on Twitter, updated in real time. As I write this, FACE WITH TEARS OF JOY (U+1F602) is the most popular emoji on Twitter, with more than 3,313,667,315 recorded occurrences.

Soapbox

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 multinational 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.

What Is “Plain Text”?

For anyone who deals with 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 nontext 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](#), which is part of the toolchain of O’Reilly’s excellent [Atlas book publishing platform](#). 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. Not all characters I wanted to show were available in the fonts used to render the book. That’s why I had to use images instead of listings in several examples in this chapter. On the other hand, the Ubuntu and macOS terminals display most Unicode text very well—including the Japanese characters for the word “mojibake”: 文字化け.

How Are str Code Points Represented in RAM?

The official Python docs avoid the issue of how the code points of a `str` are stored in memory. It is really 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.

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, two or four 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.

However, we can always count on Armin Ronacher to find problems in Python 3. He explained to me why that was not such a great idea in practice: it takes a single RAT (U+1F400) to inflate an otherwise all-ASCII text into a memory-hogging array using four bytes per character, when one byte would suffice for each character except the RAT. In addition, because of all the ways Unicode characters combine, the ability to quickly retrieve an arbitrary character by position is overrated—and extracting arbitrary slices from Unicode text is naïve at best, and often wrong, producing mojibake. As emojis become more popular, these problems will only get worse.