

COURSE

Text Encoding and Unicode

10 Chapters

Philip Osztromok

Generated with Claude

TEXT ENCODING AND UNICODE

Contents

Chapter 1 Characters and Bytes 2

Chapter 2 Inside UTF-8 8

Chapter 3 Legacy Encodings 15

Chapter 4 UTF-16 and Friends 21

Chapter 5 Normalization 28

Chapter 6 Grapheme Clusters 35

Chapter 7 Files and the Web 43

Chapter 8 Databases 51

Chapter 9 Security 59

Chapter 10 Debugging Encodings 67

Text Encoding and Unicode

Course 1 · Chapter 1 · Characters and Bytes

Every programmer meets the same bug sooner or later: a name like "café" turns into "cafÃ©", a string is "four characters long" in one tool and "five" in another, or a database column rejects an emoji. All of these come from one misunderstanding: **text is not bytes**. This chapter separates the layers (characters, numbers, bytes) and shows that "how long is this string?" has at least three different answers. Later chapters build on this map.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. Facts about Unicode and UTF-8 come from the Wikipedia articles on each (read in part: see "Not Yet Verified" at the end). Anything not checked against a source is marked "(check)".

Computers Only Store Numbers

A computer stores numbers, so to store text we agree a table that gives each character a number. An early and famous table is **ASCII**, which numbers 128 characters (0 to 127): unaccented English letters, digits, punctuation and some control codes. The capital letter A is 65. That works for English and nothing else, so many incompatible tables were invented for other languages. **Unicode** was created to replace them with one table for all writing systems. Wikipedia describes it as assigning "a unique number called a code point to each character".

A **code point** is just that number, written in hexadecimal with a `U+` prefix and at least four digits: A is `U+0041`, the euro sign is `U+20AC`. Python gives you the number with `ord()` and goes back with `chr()`:

```
for ch in "Aé€語":
    print(f"{ch!r:6} ord={ord(ch):<6} U+{ord(ch):04X}  utf-8 bytes={len(ch.encode('utf-8

print(chr(0x41), chr(0x20AC), chr(0x1F600))
```

```
'A'    ord=65     U+0041  utf-8 bytes=1
'é'    ord=233    U+00E9  utf-8 bytes=2
'€'    ord=8364   U+20AC  utf-8 bytes=3
```

```
  '語'    ord=35486  U+8A9E  utf-8 bytes=3
  A €
```

Notice what is *not* in the output of the first line: no bytes. The code point 8364 for the euro sign is an abstract number. How many bytes it takes on disk depends on a second decision, the **encoding**.

Code Space: How Many Numbers Are There?

Unicode's code space runs from `U+0000` to `U+10FFFF`, which is 1,114,112 numbers, split into 17 **planes** of 65,536. Python reports the top of the range:

```
import sys
print(sys.maxunicode, sys.maxunicode + 1)
```

```
1114111 1114112
```

Not every number is a valid character: Wikipedia says 1,112,064 of them are valid code points (the rest are surrogates, reserved for a different encoding, UTF-16, which comes later in the course). Most of the valid ones are still unassigned. Wikipedia gives the number of assigned characters in the newest version as 172,808, counting graphic and format characters only (check: this figure changes with each Unicode release).

Encoding: From Code Points to Bytes

An **encoding** is a rule for turning code points into bytes and back. The one that won on the web is **UTF-8**, designed by Ken Thompson and Rob Pike in September 1992. It uses one to four bytes per code point:

CODE POINT RANGE	UTF-8 BYTES	TYPICAL CONTENT
U+0000 to U+007F	1	ASCII: English letters, digits, punctuation
U+0080 to U+07FF	2	Accented Latin letters, Greek, Cyrillic, Hebrew, Arabic (check the exact scripts)

CODE POINT RANGE	UTF-8 BYTES	TYPICAL CONTENT
U+0800 to U+FFFF	3	Most CJK characters, the euro sign
U+10000 to U+10FFFF	4	Emoji and rarer historic scripts

The ranges match the byte counts you saw in the first program: A takes 1, é takes 2, € and 語 take 3. Because the first 128 values encode exactly like ASCII, a pure-ASCII file is already valid UTF-8, which is a large part of why UTF-8 spread so easily. Wikipedia reports that about 99.1% of surveyed websites use it (W3Techs, as quoted there).

Three Ways to Count One String

Now the trap. A reader sees "characters", but a program can count three different things:

UNIT	WHAT IT IS	PYTHON
Bytes	What is stored or sent, after an encoding is chosen	`len(s.encode("utf-8"))`
Code points	The Unicode numbers in the string	`len(s)`
Grapheme clusters	What a reader perceives as one character	Not built in (needs a library or the Unicode rules)

The letter "é" can be written two ways. One is a single code point, `U+00E9`. The other is the letter e followed by `U+0301`, a *combining acute accent* that attaches to the letter before it. They look identical and are different strings:

```
a = "\u00e9"
b = "e\u0301"
print(a, b, a == b, len(a), len(b))
print(len(a.encode("utf-8")), len(b.encode("utf-8")))
```

```
é é False 1 2
2 3
```

Two strings that look the same compare unequal, and one is longer. (Fixing this is called *normalization*, a later chapter.) The same effect is bigger with emoji, which are often several code points joined together:

```
flag = "\U0001F1EC\U0001F1E7"
print(flag, len(flag), [f"U+{ord(c):X}" for c in flag])

fam = "\U0001F468\u200D\U0001F469\u200D\U0001F467"
print(len(fam), len(fam.encode("utf-8")))
```

```
GB 2 ['U+1F1EC', 'U+1F1E7']
5 18
```

A flag is two "regional indicator" code points that a font draws as one picture. A family emoji is three person emoji glued by two invisible `U+200D` (zero-width joiner) code points: 5 code points, 18 bytes, one picture on screen (how it is drawn depends on your font, check). Counting "characters" for a username limit or a text box is therefore a design decision, not a function call.

> Common mistake
> Treating `len()` as "the number of characters a person sees". In Python 3 it is the number of code points. In some other languages the same call counts bytes or UTF-16 units instead (check per language; a later chapter covers this).

When Decoding Goes Wrong: Mojibake

Bytes mean nothing without knowing the encoding. If you write text as UTF-8 and read it back as Latin-1 (an old encoding that maps each byte to one character), each multi-byte character falls apart into several odd ones:

```
s = "café"
data = s.encode("utf-8")
print(data, data.decode("latin-1"))
```

```
b'caf\xc3\xa9' cafÃ©
```

The two bytes `C3 A9` are read as the two characters "Ã" and "©". This garbling is called **mojibake**, and recognising its shape ("Ã" followed by a symbol) is a useful debugging skill. The cure is never to guess: know which encoding produced the bytes, and decode with it. You can ask Python what a code point is:

```
import unicodedata
print(unicodedata.name("é"), "|", unicodedata.name("\u0301"), "|", unicodedata.category(
```

```
LATIN SMALL LETTER E WITH ACUTE | COMBINING ACUTE ACCENT | Mn
```

`Mn` is the category "Mark, nonspacing": a code point that draws on top of the previous one instead of taking its own space.

Exercises

EXERCISE 1

Predict, from the table above, how many UTF-8 bytes each of these needs: A, ß, Ω, €, 語 and 🎉. Then write a loop that prints each character's code point in `U+XXXX` form and its byte count, and compare with your predictions.

View solution

EXERCISE 2

Write `describe(s)`, which prints a string's code-point count, UTF-8 byte count and each code point in hex. Use it on "cafe", "café" (one code point for é), "café" (e plus U+0301) and the Japan flag (two regional indicators, U+1F1EF and U+1F1F5). Predict the numbers first.

View solution

EXERCISE 3

Encode "naïve café" as UTF-8, decode the bytes wrongly as Latin-1 to make mojibake, then reverse the damage and check you recovered the original. Also find the exact error Python raises when you decode those bytes as ASCII.

View solution

CHAPTER 1 QUICK REFERENCE

- Text is stored as numbers: a **code point** (`U+0041`) is Unicode's number for a character; an **encoding** turns code points into bytes
- Code space is `U+0000` to `U+10FFFF` (1,114,112 numbers, 17 planes)
- UTF-8 uses 1 byte for U+0000-007F (same as ASCII), 2 for up to U+07FF, 3 for up to U+FFFF, 4 above
- A string has three sizes: bytes, code points, and what the reader sees (grapheme clusters); Python's `len()` gives code points
- "é" can be one code point or two, so visually equal strings can compare unequal
- Wrong decoding produces mojibake like "cafÃ©"; know the encoding, do not guess
- `ord()`, `chr()`, `str.encode()`, `bytes.decode()` and `unicodedata.name()` are your inspection tools

NOT YET VERIFIED

Checked against Wikipedia's Unicode and UTF-8 articles (each read only in part, the first 100,000 characters): the code space size, 17 planes, 1,112,064 valid code points, the assigned-character count, UTF-8 byte ranges, the designers and date, and the 99.1% web share. The ASCII description (128 characters) and the scripts listed for 2-byte UTF-8 are from general knowledge (check), as is the behaviour of `len()` in other languages. Rendering of the flag and family emoji depends on your font. Sources: en.wikipedia.org/wiki/Unicode, en.wikipedia.org/wiki/UTF-8.

COMING NEXT

Text Encoding and Unicode 2 goes below the surface of UTF-8: how the bits of each byte are laid out, why it can recover from damage, and how to write an encoder and decoder yourself.

Text Encoding and Unicode

Course 1 · Chapter 2 · Inside UTF-8

Text Encoding and Unicode 1 showed that UTF-8 turns each code point into one to four bytes. This chapter opens the box. The rule is small enough to fit in one table, and once you see it you can write an encoder and a decoder in about thirty lines, understand why UTF-8 survives being cut in the wrong place, and see why decoders must *refuse* certain byte sequences.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. The bit patterns and validity rules were checked against RFC 3629 and the Wikipedia article on UTF-8 (the Wikipedia article was read only in part: see "Not Yet Verified" at the end). Anything else is marked " (check)".

The Whole Rule in One Table

UTF-8 puts the code point's bits into a fixed pattern. The first byte says how long the character is; every following byte is marked as a continuation. This is the table from RFC 3629, section 3:

CODE POINT RANGE	BYTE PATTERN (X = CODE POINT BITS)	FREE BITS
U+0000 to U+007F	`0xxxxxxx`	7
U+0080 to U+07FF	`110xxxxx 10xxxxxx`	11
U+0800 to U+FFFF	`1110xxxx 10xxxxxx 10xxxxxx`	16
U+10000 to U+10FFFF	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	21

Read the leading bits of a byte and you know its role:

- `0...` is a complete one-byte (ASCII) character.
- `10...` is a **continuation byte**, never the start of a character.
- `110...`, `1110...`, `11110...` are **lead bytes** for sequences of 2, 3 and 4 bytes. The number of leading 1s is the length.

Worked Example: the Euro Sign

U+20AC is in the three-byte range. Written in 16 bits it is `0010 0000 1010 1100`. Split it 4 + 6 + 6 and drop the pieces into the pattern:

```
0010 000010 101100          the 16 bits of U+20AC, split 4 + 6 + 6
1110|0010  10|000010  10|101100
11100010 10000010 10101100  =  E2 82 AC
```

Python agrees, and shows the same layout for one character of each length:

```
for ch in "A\u00e9\u20ac\U0001F600":
    bits = " ".join(f"{b:08b}" for b in ch.encode("utf-8"))
    print(f"U+{ord(ch):04X}  {bits}")
```

```
U+0041  01000001
U+00E9  11000011 10101001
U+20AC  11100010 10000010 10101100
U+1F600  11110000 10011111 10011000 10000000
```

Writing an Encoder

The table translates directly into shifts and masks. `cp >> 6` drops the low six bits; `cp & 0x3F` keeps only them; `0x80 |` sets the continuation marker. The function also refuses numbers that are not valid characters (surrogates and values above `U+10FFFF`), a rule covered below.

```
def encode_cp(cp):
    if cp < 0 or cp > 0x10FFFF or 0xD800 <= cp <= 0xDFFF:
        raise ValueError(f"not a valid scalar value: {cp:#x}")
    if cp < 0x80:
        return bytes([cp])
    if cp < 0x800:
        return bytes([0xC0 | cp >> 6, 0x80 | cp & 0x3F])
    if cp < 0x10000:
        return bytes([0xE0 | cp >> 12, 0x80 | cp >> 6 & 0x3F, 0x80 | cp & 0x3F])
    return bytes([0xF0 | cp >> 18, 0x80 | cp >> 12 & 0x3F,
                  0x80 | cp >> 6 & 0x3F, 0x80 | cp & 0x3F])

def encode(s):
    return b"".join(encode_cp(ord(c)) for c in s)
```

```
text = "Hi \u00e9 \u20ac \u8a9e \U0001F600"
mine = encode(text)
print(mine == text.encode("utf-8"), len(mine))
bad = 0
for cp in range(0x110000):
    if 0xD800 <= cp <= 0xDFFF:
        continue
    if encode_cp(cp) != chr(cp).encode("utf-8"):
        bad += 1
print("mismatches over every valid code point:", bad)
```

```
True 18
mismatches over every valid code point: 0
```

The second half is the strongest test available: it compares the hand-built encoder with Python's for every one of the 1,112,064 valid code points, and finds no difference. (Operator precedence matters in the shifts: in Python `>>` binds tighter than `&`, which binds tighter than `|`, so the expressions need no brackets, but brackets are kinder to readers.)

Self-Synchronising: Recovering from Damage

Because continuation bytes always match `10xxxxxx` and no lead byte or ASCII byte does, you can drop into the middle of a UTF-8 stream and find the next character boundary: skip forward past continuation bytes. Equally, you can find where the current character began by backing up, at most three bytes (Wikipedia). Counting the non-continuation bytes counts the code points without decoding anything:

```
def is_continuation(b):
    return b & 0xC0 == 0x80

def char_starts(data):
    return [i for i, b in enumerate(data) if not is_continuation(b)]

data = "a\u00e9\u20ac\U0001F600z".encode("utf-8")
print(data.hex(" "))
print("boundaries:", char_starts(data))
print("chars:", len(char_starts(data)))

cut = data[2:]                      # start in the middle of a character
print("cut at byte 2:", cut.hex(" "))
```

```
print("starts in cut:", char_starts(cut))
print(cut.decode("utf-8", errors="replace"))
```

```
61 c3 a9 e2 82 ac f0 9f 98 80 7a
boundaries: [0, 1, 3, 6, 10]
chars: 5
cut at byte 2: a9 e2 82 ac f0 9f 98 80 7a
starts in cut: [1, 4, 8]
🤪z
```

Starting in the middle of "é" damages only that one character (it becomes the replacement character `U+FFFD`); the decoder resynchronises at the very next lead byte and the rest is intact. Older multi-byte encodings were not so forgiving (check: this comparison is from general knowledge). The same property is why a practical text editor can jump to an arbitrary byte offset in a big file.

What a Decoder Must Refuse

The table has a subtle gap: the character U+003F could in principle be written in two bytes (`C0 BF`) by filling the two-byte pattern with leading zeros. That is an **overlong encoding**. The standard allows exactly one way to write each character, and Wikipedia explains the risk: overlong forms can "bypass other security validations", for example a filter that looks for the byte for "/" would miss a two-byte spelling of it. RFC 3629 requires implementations to protect against decoding such sequences; its example is a naive decoder turning `C0 80` into U+0000.

Three more things are invalid:

- **Surrogates** U+D800 to U+DFFF, reserved for UTF-16 (a later chapter). Their would-be encoding starts `ED A0` to `ED BF`.
- **Values above U+10FFFF**, such as the sequence `F4 90 80 80`. Earlier standards allowed up to six bytes and 31 bits; RFC 3629 (2003) cut UTF-8 back to four bytes to match the range UTF-16 can reach.
- **Malformed structure**: a continuation byte with no lead byte, or a lead byte not followed by enough continuation bytes.

That also explains the byte values that never appear in valid UTF-8: `C0` and `C1` could only start overlong forms, and `F5` to `FF` could only lead to values past U+10FFFF. Python's own decoder rejects all of these:

```
tests = {
    "overlong slash": bytes([0xC0, 0xAF]),
    "overlong NUL": bytes([0xC0, 0x80]),
    "surrogate": bytes([0xED, 0xA0, 0x80]),
    "too big": bytes([0xF4, 0x90, 0x80, 0x80]),
    "lone continuation": bytes([0x80]),
    "truncated": bytes([0xE2, 0x82]),
}
for name, raw in tests.items():
    try:
        raw.decode("utf-8")
        print(f"{name:18} decoded?!")
    except UnicodeDecodeError as e:
        print(f"{name:18} {e.reason}")
```

```
overlong slash     invalid start byte
overlong NUL       invalid start byte
surrogate          invalid continuation byte
too big            invalid continuation byte
lone continuation  invalid start byte
truncated          unexpected end of data
```

The wording is Python's own; other decoders word these differently. Notice that the surrogate case is reported as an "invalid continuation byte" rather than "surrogate": Python rejects `ED A0` because RFC 3629 limits the second byte after `ED` to `80`-`9F`. The RFC's other restricted lead bytes work the same way: `E0` needs `A0`-`BF`, `F0` needs `90`-`BF`, `F4` needs `80`-`8F`.

> Common mistake
> Writing a decoder that only checks the lead byte, or a validator that runs *after* decoding. Check on the bytes, and decode strictly (no overlongs, no surrogates), before any security check such as looking for a forbidden path or tag. Wikipedia notes that the surrogate rule is often ignored in practice, for example in Windows file names (check the details before relying on this).

Exercises

EXERCISE 1

Encode U+20AC by hand using the table: write its 16 bits, split them 4 + 6 + 6, add the pattern bits, and give the hex bytes. Then reproduce it in Python with shifts and masks only (no `.encode()` until the final comparison).

View solution

EXERCISE 2

Write a strict `decode(data)` that returns a string and raises `ValueError` for an invalid start byte, a truncated sequence, a bad continuation byte, an overlong form, a surrogate, and a value above U+10FFFF. Check it agrees with Python on valid text and rejects the five bad inputs from this chapter.

View solution

EXERCISE 3

A database column holds at most N bytes. Write `safe_truncate(data, limit)` that cuts UTF-8 bytes to at most `limit` bytes without ever splitting a character, using the continuation-byte test. Compare it with a naive slice on a string containing a two-byte, a three-byte and a four-byte character.

View solution

CHAPTER 2 QUICK REFERENCE

- Patterns: `0xxxxxxx`, `110xxxxx 10xxxxxx`, `1110xxxx 10xxxxxx 10xxxxxx`, `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
- The count of leading 1s in the first byte is the length; `10xxxxxx` is always a continuation byte
- Encode with shifts and masks: `0x80 | (cp >> n) & 0x3F` for each continuation byte
- Self-synchronising: skip or back up over bytes where `b & 0xC0 == 0x80` to find a boundary (at most three bytes back)
- Invalid: overlong forms, surrogates U+D800-DFFF, values above U+10FFFF, stray or missing continuation bytes
- Never valid as bytes: `C0`, `C1`, `F5` to `FF`
- RFC 3629 limits to four bytes; second-byte limits after `E0`, `ED`, `F0`, `F4` enforce the rules above

NOT YET VERIFIED

Checked against RFC 3629 (the octet table, the second-byte restrictions after E0, ED, F0 and F4, the surrogate ban, the overlong warning and the C0 80 example) and the Wikipedia UTF-8 article (overlong risk, back up at most three bytes, the six-byte limit before RFC 3629, the Windows file name remark). The Wikipedia article was read only in part (its first 100,000 of about 137,000 characters). The RFC page text was summarised by a fetch tool rather than read line by line. The remarks on older multi-byte encodings and the exact error wording of other decoders are from general knowledge (check). Sources: rfc-editor.org/rfc/rfc3629, en.wikipedia.org/wiki/UTF-8.

COMING NEXT

Text Encoding and Unicode 3 looks at the encodings UTF-8 replaced: ASCII, Latin-1 and the Windows code pages, and why they leave mojibake behind.

Text Encoding and Unicode

Course 1 · Chapter 3 · Legacy Encodings

UTF-8 did not arrive on an empty field. For decades, text meant one byte per character, and every region and vendor filled the spare half of the byte differently. Those older encodings are still around in old files, databases, email and web pages, and they are the source of most garbled text you will meet. This chapter explains the main ones: ASCII, Latin-1, and Windows-1252.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. Facts about ASCII, ISO-8859-1 and Windows-1252 come from their Wikipedia articles, each read only in part (see "Not Yet Verified" at the end). Details not checked against a source are marked "(check)".

ASCII: Seven Bits

ASCII was first published in 1963 as ASA X3.4-1963 (Wikipedia). It uses 7 bits and defines 128 codes, numbered 0 to 127: 33 control codes (the first 32 plus DEL at 127) and 95 printable characters. Space is 32 (`0x20`), the digit 0 is 48 (`0x30`), A is 65 (`0x41`) and a is 97 (`0x61`).

The designers placed each lowercase letter exactly `0x20` above its capital, so the two differ in a single bit (bit 5). That lets you change case with one XOR. ASCII also cannot represent anything outside basic English text, and Python refuses rather than guesses:

```
print(sum(1 for c in range(128) if chr(c).isprintable()), "printable of 128")
print(ord("A"), ord("a"), ord("a") - ord("A"), 0x20)
print(chr(ord("g") ^ 0x20), chr(ord("G") ^ 0x20))
print(f"{ord('A'):08b}")
print(f"{ord('a'):08b}")
try:
    "caf\u00e9".encode("ascii")
except UnicodeEncodeError as e:
    print(e)
print("caf\u00e9".encode("ascii", errors="replace"))
print("caf\u00e9".encode("ascii", errors="backslashreplace"))
print("caf\u00e9".encode("ascii", errors="xmlcharrefreplace"))
```

```
95 printable of 128
65 97 32 32
G g
01000001
01100001
'ascii' codec can't encode character '\xe9' in position 3: ordinal not in range(128)
b'caf?'
b'caf\\xe9'
b'caf&#233;'
```

The last three lines show Python's error handlers: `replace` silently swaps in a question mark (data is lost), `backslashreplace` keeps a readable escape, and `xmlcharrefreplace` writes an HTML numeric entity. The default, `strict`, raises an error, which is usually what you want. (Output is plain bytes, so the entity `é` shown above is literally the five characters `é` in the byte string.)

The Eighth Bit: Everyone Filled It Differently

A byte has 8 bits, so ASCII leaves 128 spare values (128 to 255). Each region and vendor filled them with what its users needed, producing many incompatible **code pages**. The same byte therefore meant different things depending on which table the reader used.

ISO-8859-1, called **Latin-1**, covers Western European languages. Its printable upper half runs from `0xA0` to `0xFF`. Its special property: every byte value is equal to the Unicode code point with the same number (Unicode's first 256 code points were chosen to match), which Python confirms below. Here is the single byte `0xE9` and the byte `0xA4` under six code pages:

```python
same = all(bytes([b]).decode("latin-1") == chr(b) for b in range(256))
print("latin-1 byte value == code point for all 256:", same)
for name in ["latin-1", "cp1252", "iso8859-15", "iso8859-7", "cp437", "koi8-r"]:
    print(f"{name:11} 0xE9 -> {bytes([0xE9]).decode(name)}   0xA4 -> {bytes([0xA4]).deco
```

```
latin-1 byte value == code point for all 256: True
latin-1     0xE9 -> é   0xA4 -> '¤'
cp1252      0xE9 -> é   0xA4 -> '¤'
iso8859-15  0xE9 -> é   0xA4 -> '€'
iso8859-7   0xE9 -> ι   0xA4 -> '€'
```

```
cp437       0xE9 -> Θ   0xA4 -> 'ñ'
koi8-r      0xE9 -> И   0xA4 -> '╓'
```

One byte, six tables, four different letters for 0xE9: é, ι (Greek iota), Θ and И. Nothing in the byte tells you which table was meant; that information must travel with the data (a later chapter covers how files and web pages declare it) or be guessed. The name `cp437` is the original IBM PC character set (check), `koi8-r` is a Russian encoding, and `iso8859-15` is a revision of Latin-1 that includes the euro sign. That `iso8859-7` (Greek) shows a euro at `0xA4` reflects a later revision of that standard in Python's table (check).

Windows-1252: The Latin-1 Lookalike

ISO-8859-1 reserves `0x80` to `0x9F` for C1 control codes, which are almost never used. It also has no euro sign (the standard dates from 1987, before the euro was presented in 1996, Wikipedia), no curly quotes and no dashes. Microsoft's **Windows-1252** matches Latin-1 everywhere except in that block, where it puts printable characters: euro sign, curly quotes, dashes, the trademark sign. It is so common that HTML5 browsers treat the label "iso-8859-1" as Windows-1252 (Wikipedia). Python shows the difference exactly:

```
diff = [b for b in range(256) if bytes([b]).decode("latin-1") != bytes([b]).decode("cp12
print(len(diff), "bytes differ; range", hex(min(diff)), "to", hex(max(diff)))
undefined = []
for b in range(0x80, 0xA0):
    try:
        bytes([b]).decode("cp1252")
    except UnicodeDecodeError:
        undefined.append(hex(b))
print("undefined in Python's cp1252:", undefined)
print(bytes([0x80]).decode("cp1252"), bytes([0x92]).decode("cp1252"), repr(bytes([0x92])
print("\u20ac".encode("cp1252"), "\u20ac".encode("iso8859-15"))
try:
    "\u20ac".encode("latin-1")
except UnicodeEncodeError as e:
    print(e)
```

```
32 bytes differ; range 0x80 to 0x9f
undefined in Python's cp1252: ['0x81', '0x8d', '0x8f', '0x90', '0x9d']
€ ’ '\x92'
```

```
b'\x80' b'\xa4'
'latin-1' codec can't encode character '\u20ac' in position 0: ordinal not in range(256)
```

All 32 bytes of the C1 block differ, and only there. Five of them (`0x81`, `0x8D`, `0x8F`, `0x90`, `0x9D`) are undefined in Windows-1252: Wikipedia says the same five are unused, and notes that some Windows APIs map them to the C1 controls anyway, a behaviour Python's strict codec does not copy. The euro sign is byte `0x80` in Windows-1252 but `0xA4` in ISO-8859-15. This is why a text pasted from a word processor, full of curly quotes, shows odd symbols if read as the wrong one of the two.

Why Mojibake Has a Shape

Last chapter ended with UTF-8 bytes read as Latin-1. The most common modern accident is the same mistake with Windows-1252 as the wrong reader. UTF-8 bytes for accented letters and curly punctuation start with `0xC3`, `0xE2` and similar, which the old tables display as "Ã" and "â", followed by symbols. Recognise that shape and you know the cause: UTF-8 text decoded as a legacy encoding.

```
original = "It\u2019s caf\u00e9"
raw = original.encode("utf-8")
garbled = raw.decode("cp1252")
print(garbled)
print(garbled.encode("cp1252").decode("utf-8") == original)

quote = "\u201d".encode("utf-8")          # right double quote
print(quote.hex(" "))
try:
    quote.decode("cp1252")
except UnicodeDecodeError as e:
    print(e)
print(quote.decode("latin-1").encode("latin-1").decode("utf-8"))
```

```
Itâ€™s cafÃ©
True
e2 80 9d
'charmap' codec can't decode byte 0x9d in position 2: character maps to <undefined>
”
```

The first half is a clean repair: encode the garbled text back with the same wrong encoding to recover the original bytes, then decode correctly. The second half shows where it breaks: the right double quote is the bytes `E2 80 9D`, and `9D` is one of Windows-1252's undefined bytes, so the strict decode fails. Latin-1, which defines all 256 bytes, never fails and can always be reversed. Whether a damaged copy can be fixed therefore depends on which wrong decoder was used, and on whether anything was replaced by `?` or `U+FFFD` along the way (then the data is gone). Libraries exist that automate such repairs (check).

> Common mistake
> Treating "it decoded without an error" as proof that the encoding is right. Latin-1 decodes any byte string without an error, so a successful decode proves nothing. Exercise 3 shows this.

Exercises

EXERCISE 1

Write `flip_case(s)` that swaps the case of ASCII letters using XOR with `0x20` and leaves everything else alone. Check it against `str.swapcase` for all 128 ASCII characters, then try it on "café" and say why the é is untouched.

View solution

EXERCISE 2

Write `compare(b)` that decodes one byte as Latin-1 and as Windows-1252, returning "(undefined)" when the second fails. Print it for 0x80, 0x85, 0x92, 0x93, 0x99, 0x9D, 0xA3 and 0xE9, then count how many of the 256 byte values differ and confirm they all lie in 0x80 to 0x9F.

View solution

EXERCISE 3

Write `which_work(raw, names)` returning the encodings in a list that decode the bytes without error. Try it on plain English, UTF-8 text with an accent, Latin-1 text with an accent, and a euro sign in Windows-1252. What does the result tell you about guessing an encoding from "no error"?

View solution

CHAPTER 3 QUICK REFERENCE

- ASCII: 7 bits, 128 codes (33 control, 95 printable); A is 65, a is 97, and case differs by one bit (`0x20`)
- Bytes 128 to 255 were assigned differently by each region and vendor: the same byte, different letters
- Latin-1 (ISO-8859-1): byte value equals Unicode code point for all 256 values; 0x80-0x9F are controls; no euro sign
- Windows-1252: Latin-1 except 0x80-0x9F, which hold printable characters (euro at 0x80, curly quotes); five bytes undefined
- HTML5 treats the "iso-8859-1" label as Windows-1252
- Mojibake like "Ã©" or "â€™" usually means UTF-8 bytes read by a legacy decoder; reverse it by encoding with the wrong codec and decoding as UTF-8
- Latin-1 never raises a decode error, so "no error" does not prove an encoding is right
- Encode error handlers: `strict` (default), `replace`, `backslashreplace`, `xmlcharrefreplace`

NOT YET VERIFIED

Checked against Wikipedia's ASCII, ISO/IEC 8859-1 and Windows-1252 articles: the 1963 publication, 7 bits and 128 codes, the 33/95 split, the one-bit case difference, Latin-1's relationship to Unicode, the missing euro sign and the reason, the HTML5 label behaviour, the euro at 0x80 and the five undefined Windows-1252 bytes. The ASCII and ISO-8859-1 articles were read only in part (first 100,000 of about 170,000 and 126,000 characters); the Windows-1252 article was summarised by a fetch tool. What the names `cp437`, `koi8-r` and `iso8859-15` refer to, the Greek euro revision, how Python's strict codec compares with Windows APIs beyond the Wikipedia remark, and the existence of automatic repair libraries are from general knowledge (check). Sources: en.wikipedia.org/wiki/ASCII, en.wikipedia.org/wiki/ISO/IEC_8859-1, en.wikipedia.org/wiki/Windows-1252.

COMING NEXT

Text Encoding and Unicode 4 covers UTF-16 and UTF-32: why a second Unicode encoding exists, what surrogate pairs are, and what byte order marks are for.

Text Encoding and Unicode

Course 1 · Chapter 4 · UTF-16 and Friends

UTF-8 is not the only way to turn code points into bytes. Two other encodings of the same Unicode table are in daily use inside operating systems, programming languages and file formats: **UTF-16** and **UTF-32**. You will meet them whenever you read a file saved by certain Windows tools, work with strings in some languages, or see a mysterious pair of bytes at the start of a file. This chapter explains how they work and what those bytes are.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. Facts about UTF-16, UTF-32 and byte order marks come from their Wikipedia articles (the UTF-16 article was read only in part: see "Not Yet Verified" at the end). Details not checked against a source are marked "(check)".

Three Encodings of One Table

All three encodings can represent every code point from `U+0000` to `U+10FFFF`. They differ in the size of the unit they use:

ENCODING	UNIT	BYTES PER CODE POINT	STRENGTH
UTF-8	8 bits	1 to 4	Compact for ASCII-heavy text; no byte-order problem
UTF-16	16 bits	2 or 4	Compact for many non-Latin scripts (check: depends on the text)
UTF-32	32 bits	always 4	The Nth code point is at a fixed position (Wikipedia: code points are "directly indexed")

UTF-32 wastes space: Wikipedia notes that 11 of its 32 bits are always zero, making it close to twice the size of UTF-16 and up to four times the size of UTF-8, depending on how many ASCII characters the text has. Here is the same four-character string in each, and the cost per character:

```
s = "A\u00e9\u20ac\U00010437"
for enc in ["utf-8", "utf-16-le", "utf-32-le"]:
    print(f"{enc:10} {len(s.encode(enc)):2} bytes")
for ch in s:
    print(f"U+{ord(ch):05X}  utf-8 {len(ch.encode('utf-8'))}  utf-16 {len(ch.encode('utf
```

```
utf-8      10 bytes
utf-16-le  10 bytes
utf-32-le  16 bytes
U+00041  utf-8 1  utf-16 2  utf-32 4
U+000E9  utf-8 2  utf-16 2  utf-32 4
U+020AC  utf-8 3  utf-16 2  utf-32 4
U+10437  utf-8 4  utf-16 4  utf-32 4
```

The `-le` suffix means little-endian byte order, explained below. U+10437 is a letter of the Deseret alphabet, chosen because it sits above U+FFFF (check: the name of the script is from general knowledge) and is also the worked example in Wikipedia's UTF-16 article.

Surrogate Pairs

A 16-bit unit can hold only 65,536 values: the range `U+0000` to `U+FFFF`, called the **Basic Multilingual Plane** (check the term; it is plane 0 from the previous chapter's 17 planes). The early 16-bit encoding, **UCS-2**, stopped there, and Wikipedia calls it obsolete. To reach the other 16 planes UTF-16 uses **surrogate pairs**: two 16-bit units that together stand for one code point. For a code point from `U+10000` up:

1. Subtract `0x10000`, leaving a 20-bit number.
2. Add the top 10 bits to `0xD800`: this is the **high surrogate** (range `D800` to `DBFF`).
3. Add the bottom 10 bits to `0xDC00`: this is the **low surrogate** (range `DC00` to `DFFF`).

This is why `U+D800` to `U+DFFF` are not assignable characters, and why UTF-8 (previous chapter) must reject their encodings. In Python:

```
def to_surrogates(cp):
    v = cp - 0x10000
    return 0xD800 + (v >> 10), 0xDC00 + (v & 0x3FF)
```

```
def from_surrogates(hi, lo):
    return 0x10000 + ((hi - 0xD800) << 10) + (lo - 0xDC00)

hi, lo = to_surrogates(0x10437)
print(f"{hi:04X} {lo:04X}", f"{from_surrogates(hi, lo):X}")
print("\U00010437".encode("utf-16-be").hex(" "))
print("\U00010437".encode("utf-16-le").hex(" "))

bad = 0
for cp in range(0x10000, 0x110000):
    raw = chr(cp).encode("utf-16-be")
    if (int.from_bytes(raw[:2], "big"), int.from_bytes(raw[2:], "big")) != to_surrogates
        bad += 1
print("mismatches against Python for all 1,048,576 supplementary code points:", bad)
```

```
D801 DC37 10437
d8 01 dc 37
01 d8 37 dc
mismatches against Python for all 1,048,576 supplementary code points: 0
```

`D801 DC37` matches the worked example in Wikipedia. The two 16-bit units appear as bytes `d8 01 dc 37` in big-endian order and `01 d8 37 dc` in little-endian order, which brings us to byte order. The final loop checks the formula against Python's own encoder for every one of the 1,048,576 code points above U+FFFF.

The Length Trap

A system that stores text as UTF-16 units counts those units, not code points. Windows APIs and Java use UTF-16 internally, and JavaScript may use UCS-2 or UTF-16 (Wikipedia), so the "length" of a string with a character above U+FFFF is 2 there, not 1 (check this per language and version). Python 3 counts code points, so you can compute the UTF-16 count yourself. The same code shows what happens to a lone surrogate:

```
lone = "\ud800"
print(len(lone), hex(ord(lone)))
try:
    lone.encode("utf-8")
except UnicodeEncodeError as e:
    print(e.reason)
print(lone.encode("utf-8", errors="surrogatepass").hex(" "))
try:
    b"\x00\xd8".decode("utf-16-le")
```

```
except UnicodeDecodeError as e:
    print(e.reason)
for s in ["a", "\u00e9", "\U0001F600", "\U00010437"]:
    print(f"code points={len(s)}  utf-16 units={len(s.encode('utf-16-le')) // 2}")
```

```
1 0xd800
surrogates not allowed
ed a0 80
unexpected end of data
code points=1  utf-16 units=1
code points=1  utf-16 units=1
code points=1  utf-16 units=2
code points=1  utf-16 units=2
```

Python lets a string hold a lone surrogate (it is one "code point"), but refuses to encode it as UTF-8, because UTF-8 forbids them. The `surrogatepass` handler forces it through, producing the invalid bytes `ED A0 80` that last chapter's decoder rejects. In UTF-16 a high surrogate with nothing after it is just as broken, here reported as "unexpected end of data". A string cut between two halves of a pair is a real bug in languages that index by UTF-16 unit.

Byte Order and the Byte Order Mark

A 16-bit unit is stored in two bytes, and there are two orders: **big-endian** (most significant byte first) and **little-endian** (least significant first). UTF-8 has no such problem, because its unit is one byte. UTF-16 and UTF-32 come in both orders. A file may begin with a **byte order mark** (BOM): the code point `U+FEFF`, which Wikipedia says is formally ZERO WIDTH NO-BREAK SPACE, written in the file's own byte order. That gives these signatures:

ENCODING	BOM BYTES
UTF-8	`EF BB BF` (optional)
UTF-16 big-endian	`FE FF`
UTF-16 little-endian	`FF FE`
UTF-32 big-endian	`00 00 FE FF`
UTF-32 little-endian	`FF FE 00 00`

If a reader uses the wrong order, the BOM turns into `U+FFFE`, a "noncharacter" that should never appear in text, which is the clue that the order is wrong. The UTF-8 BOM carries no order information; Wikipedia says the Unicode Standard neither requires nor recommends it, and that some Microsoft tools treat it as required. Python names the three behaviours separately: `utf-16` and `utf-32` write a BOM and read it to choose the order, `utf-16-le` and similar have a fixed order and no BOM, and `utf-8-sig` writes or strips the UTF-8 BOM.

```
import codecs
text = "Hi"
print(text.encode("utf-16").hex(" "))
print(text.encode("utf-16-le").hex(" "), "|", text.encode("utf-16-be").hex(" "))
print(text.encode("utf-32").hex(" "))
print(text.encode("utf-8-sig").hex(" "))
raw = text.encode("utf-16-le")
print(raw.decode("utf-16-be"))
print(repr(codecs.BOM_UTF16_LE.decode("utf-16-be")))
print(repr(b"\xef\xbb\xbfHi".decode("utf-8")), repr(b"\xef\xbb\xbfHi".decode("utf-8-sig"
```

```
ff fe 48 00 69 00
48 00 69 00 | 00 48 00 69
ff fe 00 00 48 00 00 00 69 00 00 00
ef bb bf 48 69
蹦椀
'\ufffe'
'\ufeffHi' 'Hi'
```

The plain `utf-16` output begins `ff fe`, little-endian, because this machine is little-endian (Python reports it in `sys.byteorder`; on a big-endian machine it would differ, check). Reading little-endian bytes as big-endian gives the unrelated CJK-looking characters on line six; the wrong-order BOM becomes `U+FFFE`. The last line shows a classic bug: a UTF-8 file with a BOM, opened as plain `utf-8`, begins with an invisible `U+FEFF` that breaks string comparisons, headers and parsers.

> Common mistake
> Assuming a file's encoding from the BOM alone, or ignoring it. A missing BOM proves nothing (most UTF-8 files have none). A present BOM is strong evidence, but note that `FF FE 00 00` could also be a UTF-16 little-endian BOM followed by a NUL character, so check UTF-32 signatures first (Exercise 3).

Exercises

EXERCISE 1

By hand, then in Python, work out the surrogate pairs for U+1F600 and U+24B62. Show the value after subtracting 0x10000 in binary, split it into its top and bottom 10 bits, add the bases, and compare with `chr(cp).encode("utf-16-be")`.

View solution

EXERCISE 2

Write `utf16_units(s)` that returns how many 16-bit units a string takes in UTF-16 without encoding it (2 for code points above U+FFFF, otherwise 1). Check it against the encoder on six strings, including a family emoji built from several code points, and explain why it differs from `len`.

View solution

EXERCISE 3

Write `sniff(data)` that looks for a BOM and returns the encoding name and the remaining bytes (or `None` and the data unchanged). Test it on text encoded as `utf-8-sig`, `utf-16`, `utf-16-be` and `utf-32`. Why must the UTF-32 little-endian signature be tested before the UTF-16 one?

View solution

CHAPTER 4 QUICK REFERENCE

- UTF-8 uses 1-4 bytes, UTF-16 uses 2 or 4, UTF-32 always 4
- Surrogate pair: subtract 0x10000; high = 0xD800 + top 10 bits, low = 0xDC00 + bottom 10 bits
- High surrogates D800-DBFF, low surrogates DC00-DFFF; they are not characters and are illegal in UTF-8
- UCS-2 is the obsolete 16-bit-only predecessor of UTF-16
- Systems that index by UTF-16 unit report 2 for a character above U+FFFF; Python reports 1
- BOM is U+FEFF: `FE FF` / `FF FE` (UTF-16 BE/LE), `00 00 FE FF` / `FF FE 00 00` (UTF-32), `EF BB BF` (UTF-8, optional)

- Wrong byte order turns the BOM into U+FFFE; a UTF-8 BOM read as plain UTF-8 leaves an invisible U+FEFF (use `utf-8-sig`)
- Python codecs: `utf-16` and `utf-32` add and read a BOM; `-le`/`-be` variants do not

NOT YET VERIFIED

Checked against Wikipedia's UTF-16, Byte order mark and UTF-32 articles: the surrogate formula and ranges, the D801 DC37 worked example, UCS-2 being obsolete, which systems use UTF-16, the BOM byte sequences, U+FEFF's formal name, the wrong-order U+FFFE effect, the UTF-8 BOM remarks, and UTF-32's fixed width, space cost and direct indexing. The UTF-16 article was read only in part (first 100,000 of about 113,000 characters); the other two were summarised by a fetch tool. The UTF-16 article did not itself spell out the BOM bytes, so the byte values come from the BOM article and from Python's own output. The "Basic Multilingual Plane" name, the script name for U+10437, how each language counts length, and the compactness claims are from general knowledge (check). Sources: en.wikipedia.org/wiki/UTF-16, en.wikipedia.org/wiki/Byte_order_mark, en.wikipedia.org/wiki/UTF-32.

COMING NEXT

Text Encoding and Unicode 5 returns to the two-spellings problem from chapter 1 and fixes it: normalization forms, and how to compare and search text that looks the same.

Text Encoding and Unicode

Course 1 · Chapter 5 · Normalization

In *Text Encoding and Unicode 1* we met two strings that look identical but compare unequal: "é" as one code point, and "e" followed by a combining accent. That was a preview of a real problem: Unicode often gives several ways to write the same text. A login that accepts "Zoë" on one device and rejects it on another, a search that misses a word, a duplicate row in a database: these usually come from this. The cure is **normalization**, and this chapter explains what it does and when to use which form.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. Facts about normalization come from the Wikipedia article "Unicode equivalence" and from Unicode Standard Annex #15 (UAX #15), both read through a summarising tool (see "Not Yet Verified" at the end). Details not checked against a source are marked "(check)".

Two Kinds of "Same"

Unicode defines two levels of equivalence between sequences of code points:

KIND	MEANING	EXAMPLE
Canonical equivalence	Assumed to have the same appearance and meaning; may be substituted for each other	`U+00E9` and `e + U+0301`; the angstrom sign `U+212B` and `U+00C5`
Compatibility equivalence	May look different but share a meaning in some contexts; treating them alike is sometimes right and sometimes wrong	The ligature `U+FB03` and "ffi"; the superscript `U+2075` and "5"; fullwidth A and A

Canonical equivalence implies compatibility equivalence, but not the other way round (Wikipedia). The first kind is about the same text spelled two ways; the second is about different characters that are close enough for some purposes, such as searching.

The Four Normalization Forms

A normalization form converts a string into one fixed representative of its equivalence class, so that equivalent strings become identical. There are four, built from two operations: *decomposition* (split into base letter plus marks) and *composition* (join back where a single code point exists):

FORM	WHAT IT DOES
NFD	Canonical decomposition, with combining marks in a set order
NFC	Canonical decomposition, then canonical composition: the shortest, "precomposed" form
NFKD	Compatibility decomposition, with combining marks in a set order
NFKC	Compatibility decomposition, then canonical composition

The "K" stands for compatibility. Python exposes all four through `unicodedata.normalize`. Start with the problem from chapter 1:

```
import unicodedata as ud

a = "\u00e9"
b = "e\u0301"
print(a == b, len(a), len(b))
for form in ["NFC", "NFD"]:
    na, nb = ud.normalize(form, a), ud.normalize(form, b)
    print(form, na == nb, len(na), len(nb))

def same_text(x, y):
    return ud.normalize("NFC", x) == ud.normalize("NFC", y)

print(same_text(a, b))
```

```
False 1 2
NFC True 1 1
NFD True 2 2
True
```

Either form makes the two strings equal; NFC makes both one code point long and NFD makes both two. UAX #15 states that equivalent strings have "precisely the same normalized form", which is what allows plain binary comparison afterwards. It also states that each form is *idempotent*: normalizing twice changes nothing.

What Each Form Does to Awkward Characters

Run all four forms over six characters and the differences between the canonical pair (NFC, NFD) and the compatibility pair (NFKC, NFKD) become clear:

```
import unicodedata as ud

forms = ["NFC", "NFD", "NFKC", "NFKD"]
samples = {
    "angstrom sign": "\u212b",
    "A with ring": "\u00c5",
    "ffi ligature": "\ufb03",
    "superscript 5": "\u2075",
    "roman numeral IV": "\u2163",
    "fullwidth A": "\uff21",
}
for label, s in samples.items():
    row = []
    for f in forms:
        n = ud.normalize(f, s)
        row.append(" ".join(f"{ord(c):04X}" for c in n))
    print(f"{label:17} U+{ord(s):04X} | " + " | ".join(f"{f}: {r}" for f, r in zip(forms
```

```
angstrom sign     U+212B | NFC: 00C5 | NFD: 0041 030A | NFKC: 00C5 | NFKD: 0041 030A
A with ring       U+00C5 | NFC: 00C5 | NFD: 0041 030A | NFKC: 00C5 | NFKD: 0041 030A
ffi ligature      U+FB03 | NFC: FB03 | NFD: FB03 | NFKC: 0066 0066 0069 | NFKD: 0066 006
superscript 5     U+2075 | NFC: 2075 | NFD: 2075 | NFKC: 0035 | NFKD: 0035
roman numeral IV  U+2163 | NFC: 2163 | NFD: 2163 | NFKC: 0049 0056 | NFKD: 0049 0056
fullwidth A       U+FF21 | NFC: FF21 | NFD: FF21 | NFKC: 0041 | NFKD: 0041
```

Reading the table:

- The angstrom sign and the Swedish letter Å are canonically equivalent, so all forms fold the sign into `U+00C5` (NFC) or A plus `U+030A`, a combining ring (NFD).
- The ligature, the superscript, the Roman numeral and the fullwidth letter are only *compatibility* equivalents: NFC and NFD leave them alone, while NFKC and NFKD replace them with plain letters and digits. A search for "f" matches the ligature under NFKC but not under NFC (Wikipedia).

Combining Marks Have an Order

When a letter carries several marks, the order they are typed in can differ without changing the look. Each mark has a *combining class* number, and normalization sorts adjacent marks of non-zero class by that number (UAX #15), keeping the original order for marks of equal class, since those interact visually. Python reports the class with `unicodedata.combining`:

```
import unicodedata as ud

s = "e\u0323\u0302"          # e + dot below + circumflex
t = "e\u0302\u0323"          # e + circumflex + dot below
print(s == t, [ud.combining(c) for c in s], [ud.combining(c) for c in t])
print(ud.normalize("NFD", s) == ud.normalize("NFD", t))
print(ud.normalize("NFC", s) == ud.normalize("NFC", t), [f"{ord(c):04X}" for c in ud.nor

u = "\u1ebf"                # e with circumflex and acute
d = ud.normalize("NFD", u)
print([f"{ord(c):04X}" for c in d], [ud.combining(c) for c in d])
print(ud.normalize("NFC", "e\u0302\u0301") == u, ud.normalize("NFC", "e\u0301\u0302") ==
print(ud.normalize("NFC", ud.normalize("NFC", s)) == ud.normalize("NFC", s))
```

```
False [0, 220, 230] [0, 230, 220]
True
True ['1EC7']
['0065', '0302', '0301'] [0, 230, 230]
True False
True
```

The dot below has class 220 and the circumflex class 230, so both orders sort to the same sequence, and under NFC both become the single code point `U+1EC7`. In the second part the acute and circumflex share class 230, so their typed order is kept: "e, circumflex, acute" composes to `U+1EBF`, but "e, acute, circumflex" does not. The last line demonstrates idempotence.

Which Form, When?

- **NFC** is the usual choice for storing and exchanging text and for comparing strings. It is compact, and it keeps compatibility characters as they were. (I believe the W3C recommends

NFC for web content: check, neither source I read said so.)

- **NFD** is useful as a working form when you want to take apart letters and marks, for example to strip accents (Exercise 3).
- **NFKC / NFKD** are for *matching*, such as search or identifier comparison, where folding ligatures and fullwidth letters is wanted. UAX #15 warns they "must not be blindly applied to arbitrary text", because they discard distinctions.

The loss is real. Here a user-name key is built with NFKC followed by `casefold()` (a stronger lowercase), and the same call is shown ruining a sentence:

```
import unicodedata as ud

names = ["Zo\u00eb", "Zoe\u0308", "ZO\u00cb"]
print(len(set(names)), len(set(ud.normalize("NFC", n) for n in names)))
print(len(set(ud.normalize("NFC", n).casefold() for n in names)))

text = "The \ufb01nal \uff21\uff22\uff23 costs 2\u00b2 \u2163"
print(text)
print(ud.normalize("NFC", text))
print(ud.normalize("NFKC", text))
print("final" in ud.normalize("NFC", text), "final" in ud.normalize("NFKC", text))

x = ud.normalize("NFC", "e")
y = ud.normalize("NFC", "\u0301")
print(len(ud.normalize("NFC", x + y)), len(x + y))
```

```
3 2
1
The final A B C costs 2² Ⅳ
The final A B C costs 2² Ⅳ
The final ABC costs 22 IV
False True
1 2
```

Three lessons from the output. First, three spellings of one name collapse to two strings after NFC (the third differs only in case) and to one after `casefold`. Second, NFKC makes "final" findable (the ligature is gone), but it also turned "2^2" into "22", a meaning change, which is why NFKC suits matching and not stored text. Third, normalized strings are not closed under concatenation (UAX #15): "e" and a lone accent are each already in NFC, but joined they are not,

because they now compose into one code point (length 1, not 2). Normalize after you join, not before.

> Common mistake
> Normalizing only one side of a comparison, or normalizing in one place and not another. Pick a form, apply it where text enters your system (input, import, API), and compare only normalized strings. Also remember that normalization does not fix case, accents you think of as "the same letter" (it will not turn "ø" into "o"), or a mix-up of encodings (that is chapter 3's problem).

Exercises

EXERCISE 1

Write `canon_eq(a, b)` and `compat_eq(a, b)` using NFD and NFKD. Test six pairs: é and e+acute, Å and the angstrom sign, the ffi ligature and "ffi", two mark orders that share a combining class, fullwidth A and A, and a and A. Predict each answer first.

View solution

EXERCISE 2

Write `key(name)` (NFKC then `casefold()`) and use it to group six user names that differ by composition, case, fullwidth letters, a missing accent and a plural. How many distinct names are there raw, and how many after keying?

View solution

EXERCISE 3

Write `strip_accents(s)`: decompose with NFD, drop characters with a non-zero combining class, and recompose with NFC. Try it on "Zoë", "crème brûlée", "Ångström", "øre", "straße" and "Łódź". Which letters does it not change, and why?

View solution

CHAPTER 5 QUICK REFERENCE

- Canonical equivalence: same text spelled differently (é vs e+accent); compatibility equivalence: similar characters (ligatures, superscripts, fullwidth)
- NFD = canonical decompose; NFC = decompose then compose; NFKD / NFKC = the same with compatibility mappings
- Equivalent strings share one normalized form; every form is idempotent
- Combining marks are sorted by combining class; marks of the same class keep their order
- Use NFC to store and compare; NFD to take letters apart; NFKC/NFKD only for matching, since they lose information ("2^2" became "22")
- Normalization is not closed under concatenation: normalize after joining
- Python: `unicodedata.normalize(form, s)`, `unicodedata.combining(c)`, `str.casefold()`

NOT YET VERIFIED

Checked, through a summarising fetch tool and not by reading line by line, against Wikipedia's "Unicode equivalence" article and UAX #15: the two kinds of equivalence, the definitions of the four forms, the angstrom, ligature, superscript and fullwidth examples, canonical ordering by combining class, idempotence, the NFKC warning, and non-closure under concatenation. Neither source, as summarised, stated which form the W3C recommends, so that remark is general knowledge (check). The specific outputs for U+2163, U+00B2 and U+1EC7 come from Python's own tables, not from the sources. The remark that casefold is "a stronger lowercase" is from general knowledge (check). Sources: en.wikipedia.org/wiki/Unicode_equivalence, unicode.org/reports/tr15/.

COMING NEXT

Text Encoding and Unicode 6 asks what a "character" is to a reader: grapheme clusters, how emoji sequences are built, and why counting characters correctly needs more than normalization.

Text Encoding and Unicode

Course 1 · Chapter 6 · Grapheme Clusters

Text Encoding and Unicode 1 showed three ways to measure a string: bytes, code points, and what a reader sees. Normalization (the previous chapter) cannot give you the third one. A family emoji, a flag, or a letter with two accents is still several code points after any normalization, yet a reader sees one thing. This chapter defines that unit, the **grapheme cluster**, shows how emoji and Hangul are built from pieces, and uses it to count, cut and reverse text without breaking it.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. The standard library has no grapheme function (check), so the examples use the third-party `regex` module (install with `pip install regex`), whose `\X` pattern matches one extended grapheme cluster. Facts come from Unicode Standard Annex #29 (UAX #29) and the Wikipedia article on emoji, both read through a summarising tool (see "Not Yet Verified" at the end). Details not checked against a source are marked "(check)".

What Is a Character, to a Reader?

UAX #29 calls what a person thinks of as one character a *user-perceived character*, and approximates it with the **grapheme cluster**. Its own example is the letter G with a grave accent: two code points, one perceived character. Grapheme clusters are the units for moving a cursor, selecting, deleting and counting "characters", and a word, line or sentence break should never fall inside one. The standard defines *extended* grapheme clusters, the recommended default, and older *legacy* ones kept for compatibility.

Many things that count as a single cluster are built from several code points, each with a job:

PIECE	CODE POINTS	EFFECT
Combining mark	e.g. `U+0301`	Attaches to the previous letter
Skin tone modifier	`U+1F3FB` to `U+1F3FF`	Follows a human emoji and sets its tone (added in Unicode 8.0, Wikipedia)
Zero-width joiner (ZWJ)	`U+200D`	Glues emoji into one picture, e.g. a family; unsupported systems show the parts side by side

PIECE	CODE POINTS	EFFECT
Regional indicators	`U+1F1E6` to `U+1F1FF`	26 letters; a pair spells a flag
Variation selector 16	`U+FE0F`	Asks for the colourful emoji look (VS15, `U+FE0E`, asks for plain text look)
Keycap sequence	digit or # or *, `U+FE0F`, `U+20E3`	Makes a keycap picture (the exact recipe is from general knowledge: check)

One Cluster, Many Code Points

```
import regex

samples = {
    "e + acute": "e\u0301",
    "thumbs up + tone": "\U0001F44D\U0001F3FD",
    "UK flag": "\U0001F1EC\U0001F1E7",
    "family": "\U0001F468\u200D\U0001F469\u200D\U0001F467",
    "red heart": "\u2764\uFE0F",
    "keycap 1": "1\uFE0F\u20E3",
    "CR LF": "\r\n",
}
for label, s in samples.items():
    clusters = regex.findall(r"\X", s)
    print(f"{label:17} code points={len(s)}  utf-8 bytes={len(s.encode('utf-8')):2}  clu
```

```
e + acute         code points=2  utf-8 bytes= 3  clusters=1
thumbs up + tone  code points=2  utf-8 bytes= 8  clusters=1
UK flag           code points=2  utf-8 bytes= 8  clusters=1
family            code points=5  utf-8 bytes=18  clusters=1
red heart         code points=2  utf-8 bytes= 6  clusters=1
keycap 1          code points=3  utf-8 bytes= 7  clusters=1
CR LF             code points=2  utf-8 bytes= 2  clusters=1
```

Every sample is one thing to a reader and one cluster to the library, while the code point and byte counts range up to 5 and 18. Even CR LF (a Windows-style line break) is one cluster, because rule GB3 forbids a break between them. Zoom in on emoji parts:

```
import regex

def count(s):
    return len(regex.findall(r"\X", s))

def report(s):
    cps = " ".join(f"U+{ord(c):04X}" for c in s)
    print(f"{len(s)} code points, {count(s)} clusters: {cps}")

report("\U0001F44D")
report("\U0001F44D\U0001F3FD")
report("\U0001F469\U0001F3FB\u200D\U0001F4BB")
report("\U0001F1E9\U0001F1EA")
report("\u2764")
report("\u2764\uFE0F")
```

```
1 code points, 1 clusters: U+1F44D
2 code points, 1 clusters: U+1F44D U+1F3FD
4 code points, 1 clusters: U+1F469 U+1F3FB U+200D U+1F4BB
2 code points, 1 clusters: U+1F1E9 U+1F1EA
1 code points, 1 clusters: U+2764
2 code points, 1 clusters: U+2764 U+FE0F
```

The third line is a tone-modified person joined by ZWJ to another emoji: four code points, one picture. The heart shows that `U+2764` alone and followed by `U+FE0F` are both one cluster, differing in how they should be drawn.

The Rules Behind the Clusters

UAX #29 gives every code point a `Grapheme_Cluster_Break` property (CR, LF, Control, Extend, ZWJ, Regional_Indicator, Prepend, SpacingMark, and the Hangul types L, V, T, LV and LVT) and then a list of rules. In plain terms:

- **GB3**: do not break between CR and LF. **GB4, GB5**: otherwise always break around controls.
- **GB6 to GB8**: keep Hangul jamo (and syllable) sequences together.
- **GB9**: do not break before Extend or ZWJ characters (this keeps a letter with its marks). **GB9a**: do not break before a SpacingMark. **GB9b**: do not break after a Prepend character.
- **GB11**: do not break inside an emoji ZWJ sequence.

- **GB12, GB13**: regional indicators pair up into flags; an odd one out is left alone.
- **GB999**: anywhere else, break.

Hangul shows why composition does not matter to the rules: a syllable written as one code point and the same syllable as three conjoining jamo are both single clusters. Flags show the odd-one-out rule:

```
import regex
import unicodedata as ud

for form in ["NFC", "NFD"]:
    s = ud.normalize(form, "\uac01")
    print(form, [f"{ord(c):04X}" for c in s], len(regex.findall(r"\X", s)))

two_flags = "\U0001F1EC\U0001F1E7\U0001F1EB\U0001F1F7"
three_ri = two_flags + "\U0001F1E9"
print(len(regex.findall(r"\X", two_flags)), len(regex.findall(r"\X", three_ri)))
```

```
NFC ['AC01'] 1
NFD ['1100', '1161', '11A8'] 1
2 3
```

Four regional indicators make two flags; a fifth makes a third cluster that is not a valid flag by itself, but is still a cluster. The Hangul syllable 각 (`U+AC01`) is the example UAX #29 itself uses.

A Small Segmenter

You do not need to write one in practice, but building a simplified one makes the rules concrete. This version treats marks, joiners, variation selectors and skin tones as "extend", glues whatever follows a ZWJ, pairs regional indicators, and keeps CR LF together. It leaves out Hangul, Prepend and SpacingMark, so it is a teaching sketch and not UAX #29:

```
import unicodedata as ud

def is_extend(c):
    cp = ord(c)
    return (ud.category(c) in ("Mn", "Me")      # nonspacing and enclosing marks
            or cp == 0x200D                      # zero width joiner (handled separately
            or 0xFE00 <= cp <= 0xFE0F            # variation selectors
```

```
            or 0x1F3FB <= cp <= 0x1F3FF)        # emoji skin tone modifiers

def is_ri(c):
    return 0x1F1E6 <= ord(c) <= 0x1F1FF

def clusters(s):
    out = []
    i = 0
    while i < len(s):
        j = i + 1
        if s[i:i + 2] == "\r\n":
            j = i + 2
        elif is_ri(s[i]) and j < len(s) and is_ri(s[j]):
            j = i + 2
        while j < len(s) and (is_extend(s[j]) or s[j - 1] == "\u200D"):
            j += 1
        out.append(s[i:j])
        i = j
    return out

text = "e\u0301 \U0001F44D\U0001F3FD \U0001F1EC\U0001F1E7 \U0001F468\u200D\U0001F469\u200
mine = clusters(text)
print(len(text), len(mine))
import regex
print(mine == regex.findall(r"\X", text))
```

```
24 13
True
```

On this sample, 24 code points make 13 clusters (seven things plus six spaces), and the sketch agrees with the library. It would not on Hangul jamo or Indic scripts, and the rule list above is

Using Clusters: Counting, Cutting, Reversing

Cutting a string at a code point index can split a cluster, leaving a stray accent, half a family, or a dangling joiner. Cut at cluster boundaries instead:

```
import regex
```

```
def graphemes(s):
    return regex.findall(r"\X", s)

def truncate(s, n):
    return "".join(graphemes(s)[:n])

s = "cafe\u0301 \U0001F468\u200D\U0001F469\u200D\U0001F467!"
print(len(s), len(graphemes(s)))
print(ascii(s[:4]), ascii(truncate(s, 4)))
print(ascii(s[:7]))
print(ascii(truncate(s, 6)))
rev = graphemes(s)[::-1]
print(ascii(s[::-1][:3]), ascii(rev[:2]))
```

```
12 7
'cafe' 'cafe\u0301'
'cafe\u0301 \U0001f468'
'cafe\u0301 \U0001f468\u200d\U0001f469\u200d\U0001f467'
'!\U0001f467\u200d' ['!', '\U0001f468\u200d\U0001f469\u200d\U0001f467']
```

(`ascii()` shows the escapes so the invisible parts can be seen.) A four-code-point slice lost the accent from "café", and a seven-code-point slice kept only the man of the family. Reversing by code points left the string starting with a stray girl and a joiner, while reversing the list of clusters kept each cluster intact. The same logic applies to character limits in forms: count clusters if you mean what a reader sees, but remember that storage limits are still in bytes or code points (a database may count differently: next chapters).

> Common mistake
> Assuming the cluster rules are fixed. New emoji sequences appear with each Unicode release, so a library built on an older Unicode version may split a newer sequence (check). Python's own `unicodedata` reports its Unicode version in `unicodedata.unidata_version`, and a library's data may be older or newer than that. Also, a cluster is a good default for "one character", not a perfect one: what a reader sees depends on the font and language.

Exercises

EXERCISE 1

Write `report(s)` printing the number of code points, the number of clusters and each code point in hex. Use it on a thumbs-up, the same thumbs-up with a skin tone, a woman with a tone joined to a laptop (use ZWJ), a flag, a plain heart and a heart with U+FE0F. Predict the cluster counts first.

View solution

EXERCISE 2

Show that composition does not change cluster counts. Take three Hangul syllables, count code points and clusters under NFC and under NFD. Then show that "a", CR LF, "b" is 3 clusters but 4 code points, and compare a letter followed by two combining marks with a combining mark followed by a letter.

View solution

EXERCISE 3

Write `truncate(s, limit)` that shortens text to at most `limit` clusters, ending with an ellipsis when it cuts, and never splitting a cluster. Test it on plain text, text with a combining accent, and text containing a family emoji. Compare with a naive slice.

View solution

CHAPTER 6 QUICK REFERENCE

- A grapheme cluster approximates a user-perceived character; use it for counting, cursor movement, selection, truncation and reversal
- Emoji parts: skin tone modifiers (U+1F3FB-1F3FF), ZWJ (U+200D), regional indicators (U+1F1E6-1F1FF, pairs make flags), variation selector U+FE0F
- Rules (UAX #29): GB3 keeps CR LF together; GB9 keeps marks and ZWJ with their base; GB11 keeps emoji ZWJ sequences; GB12/13 pair regional indicators
- Hangul syllables and conjoining jamo are single clusters either way, so normalization does not change cluster counts
- Python's standard library has no grapheme function (check); the `regex` module's `\X` matches one extended grapheme cluster
- Never slice or reverse by code point when the output is for people; cut and reverse lists of clusters

- Segmentation rules grow with each Unicode version; keep libraries current

NOT YET VERIFIED

Checked, through a summarising fetch tool and not by reading line by line, against UAX #29 (user-perceived characters, extended versus legacy clusters, the property values, rules GB3 to GB13 in plain terms, the Hangul and flag examples) and Wikipedia's emoji article (skin tone range and Unicode 8.0, ZWJ, regional indicator range, VS15 and VS16). The emoji article was read only in part (first 100,000 of about 228,000 characters). The article did not explain keycap sequences or the flag country-code mapping, so the keycap recipe is from general knowledge (check). That the standard library has no grapheme segmentation, how third-party libraries track Unicode versions, and the behaviour of the sample outputs on other library versions are also from general knowledge or from this one run (check). The simplified segmenter is my own sketch and deliberately incomplete. Sources: unicode.org/reports/tr29/, en.wikipedia.org/wiki/Emoji.

COMING NEXT

Text Encoding and Unicode 7 follows text out of the program: files, network protocols and web pages, and how each of them says (or fails to say) which encoding the bytes use.

Text Encoding and Unicode

Course 1 · Chapter 7 · Files and the Web

Inside a program, text is a sequence of code points. The moment it is saved to a file, sent over a network or put in a web page, it becomes **bytes**, and the reader at the other end needs to know which encoding to use to get the text back. This chapter walks through the main places where that information lives, or fails to live: file defaults, newline conventions, HTTP headers, HTML declarations, JSON and CSV. Most real-world mojibake starts at one of these boundaries.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 on a Windows machine, and the output shown is the real output. Two outputs depend on the machine and are marked as such. Facts about HTTP, the web platform, JSON and Python's defaults come from MDN, the WHATWG HTML Standard, RFC 8259 and PEP 686, each read through a summarising tool (see "Not Yet Verified" at the end). Details not checked against a source are marked "(check)".

Files: Always Name the Encoding

A plain text file has no field saying how it is encoded (unless it starts with a BOM, see Text Encoding and Unicode 4). When Python opens a file in text mode without an `encoding` argument, it picks a default that depends on the platform. PEP 686 describes this default as platform-dependent, and says that UTF-8 mode, which makes it consistent, becomes the default starting with Python 3.15 (check the final release status). On this Windows machine the difference is visible when the same program runs with and without UTF-8 mode:

```
import subprocess, sys

code = "import locale, os, sys; print(sys.flags.utf8_mode, locale.getencoding(), open(os
for flags in ([], ["-X", "utf8"]):
    r = subprocess.run([sys.executable, *flags, "-c", code], capture_output=True, text=T
    print(flags or "default", "->", r.stdout.strip())
```

```
default -> 0 cp1252 cp1252
['-X', 'utf8'] -> 1 cp1252 utf-8
```

Each line shows the UTF-8 mode flag, the machine's locale encoding, and the encoding a text-mode `open()` would use. **This output is specific to this machine** (a Western European Windows install): by default `open()` uses Windows-1252, so a UTF-8 file written on Linux and read here without an `encoding` argument is silently misread as in Text Encoding and Unicode 3. With UTF-8 mode on, the locale is unchanged but `open()` uses UTF-8. PEP 686 notes this default mainly affects Windows users, because most Unix systems already use a UTF-8 locale. The fix needs no flag: **always pass** `encoding`.

```
import tempfile, pathlib

d = pathlib.Path(tempfile.mkdtemp())
p = d / "note.txt"
with open(p, "w", encoding="utf-8", newline="\n") as f:
    f.write("caf\u00e9 \u20ac\n")
raw = p.read_bytes()
print(raw)
print(repr(raw.decode("utf-8")))
print(repr(raw.decode("cp1252", errors="replace")))
try:
    p.read_text(encoding="ascii")
except UnicodeDecodeError as e:
    print(e.reason)
print(ascii(p.read_text(encoding="ascii", errors="replace")))
```

```
b'caf\xc3\xa9 \xe2\x82\xac\n'
'café €\n'
'cafÃ© â,¬\n'
ordinal not in range(128)
'caf\ufffd\ufffd \ufffd\ufffd\ufffd\n'
```

The file holds ten bytes: "caf" is three, é is the two bytes `c3 a9`, then a space, then € as the three bytes `e2 82 ac`, then the line feed. Read as UTF-8 it round-trips; read as Windows-1252 it turns into mojibake; read as ASCII it fails, or with `errors="replace"` every non-ASCII *byte* becomes its own replacement character (five here for two characters: another reminder that bytes and characters differ).

Newlines Are Part of the Encoding Problem

Line endings differ too: `LF` (`\n`) on Unix-like systems and `CR LF` (`\r\n`) on Windows. Python's text mode translates between `\n` in your program and the platform's convention. The `newline` argument controls it, and it matters for byte-exact files:

```
import tempfile, pathlib

d = pathlib.Path(tempfile.mkdtemp())
p = d / "lines.txt"
with open(p, "w", encoding="utf-8", newline="\n") as f:
    f.write("a\nb\n")
print(p.read_bytes())
with open(p, "w", encoding="utf-8", newline="\r\n") as f:
    f.write("a\nb\n")
print(p.read_bytes())
with open(p, encoding="utf-8", newline="") as f:
    print(repr(f.read()))
with open(p, encoding="utf-8") as f:
    print(repr(f.read()))
```

```
b'a\nb\n'
b'a\r\nb\r\n'
'a\r\nb\r\n'
'a\nb\n'
```

With `newline=""` on reading, nothing is translated and the CR LF survives; by default, reading converts CR LF to `\n`. Python's `csv` module asks you to open files with `newline=""` so it can handle line endings itself (check the wording in the csv documentation).

HTTP: The Header Carries the Charset

An HTTP response describes its body with a `Content-Type` header, whose `charset` parameter names the encoding. MDN gives the example `text/html; charset=utf-8` and says the value is case-insensitive but lowercase is preferred. MDN's page does not say what applies when the charset is missing; for web pages the HTML rules below fill the gap. Python can parse the parameter with the standard `email` package (the header syntax is shared; this is a convenience, not a full HTTP client):

```
from email.message import Message
```

```
def charset_of(content_type):
    m = Message()
    m["Content-Type"] = content_type
    return m.get_content_charset()

for ct in ["text/html; charset=utf-8", "text/html; charset=UTF-8", 'text/plain; charset=
    print(f"{ct:36} -> {charset_of(ct)}")

body = "caf\u00e9".encode("utf-8")
print(body.decode(charset_of("text/html; charset=utf-8") or "utf-8"))
print(body.decode(charset_of('text/plain; charset="iso-8859-1"')))
```

```
text/html; charset=utf-8             -> utf-8
text/plain; charset="iso-8859-1"     -> iso-8859-1
text/html                            -> None
application/json                     -> None
café
cafÃ©
```

The parser lowercases the value, strips quotes, and returns `None` when absent. The last line shows what a wrong header does: the same bytes, decoded as the header's Latin-1, give mojibake. The server's header is the claim; the bytes are the truth; mismatch is a bug on the server side.

JSON: Settled by the Standard

JSON removed the guesswork. RFC 8259 says JSON exchanged between systems outside a closed ecosystem "MUST be encoded using UTF-8", that implementations must not add a BOM to network JSON, and that parsers may ignore one. Inside a JSON string, a character outside the BMP can be written as an escaped surrogate pair (the RFC's example is the G clef, `\uD834\uDD1E`), which is why the surrogates of Text Encoding and Unicode 4 still matter. Python's `json` escapes non-ASCII by default and can be told not to:

```
import csv, io, json

obj = {"name": "caf\u00e9", "face": "\U0001F600"}
print(json.dumps(obj))
print(json.dumps(obj, ensure_ascii=False))
print(json.loads(r'"\ud83d\ude00"') == "\U0001F600")
```

```
buf = io.BytesIO()
text = io.TextIOWrapper(buf, encoding="utf-8-sig", newline="")
csv.writer(text).writerow(["name", "caf\u00e9"])
text.flush()
print(buf.getvalue())
```

```
{"name": "caf\u00e9", "face": "\ud83d\ude00"}
{"name": "café", "face": "😀"}
True
b'\xef\xbb\xbfname,caf\xc3\xa9\r\n'
```

The first line is pure ASCII, safe in any pipeline, with the emoji as a surrogate pair; the second keeps the characters and relies on the receiver reading UTF-8. The third line shows a parser joining the pair back into one character. The last line shows a CSV written with the `utf-8-sig` codec, which starts with the BOM bytes `EF BB BF`. That BOM is a workaround: some spreadsheet programs only recognise UTF-8 when it is present (check: this is common experience, not something I verified in a source). It conflicts with the JSON rule above, so decide per format.

HTML: Declarations and a 1024-Byte Window

A browser must decide the encoding of a page from several clues. The WHATWG HTML Standard lists them in order of priority:

1. a byte order mark;
2. an explicit choice by the user;
3. the transport layer, such as the HTTP `charset`;
4. a prescan of the first bytes for a `<meta>` declaration (the standard encourages limiting it to 1024 bytes, and says authoring rules restrict declarations to that window);
5. the parent document, a previous visit, autodetection (discouraged for network resources);
6. an implementation default, which depends on the user's locale; for all locales not listed, windows-1252.

The standard recommends UTF-8 where the author can choose, and describes the locale-based defaults as a compatibility measure for legacy content. Here is a deliberately simplified version of the first, third, fourth and last of those steps (it omits the user override and the other tentative clues, and a real prescan is stricter than this regular expression):

```
import codecs
import re

META = re.compile(rb'<meta[^>]+charset\s*=\s*["\']?([A-Za-z0-9_\-]+)', re.I)

def sniff_html(data, http_charset=None):
    for bom, name in [(codecs.BOM_UTF8, "utf-8"), (codecs.BOM_UTF16_LE, "utf-16-le"), (c
        if data.startswith(bom):
            return name, "BOM"
    if http_charset:
        return http_charset, "HTTP header"
    m = META.search(data[:1024])
    if m:
        return m.group(1).decode("ascii"), "meta tag"
    return "cp1252", "default"

page = '<html><head><meta charset="utf-8"><title>caf\u00e9</title></head>'.encode("utf-8
late = b"<html><head>" + b"<!-- " + b"x" * 1100 + b" -->" + b'<meta charset="utf-8"></he
print(sniff_html(page))
print(sniff_html(page, "iso-8859-1"))
print(sniff_html(codecs.BOM_UTF8 + page, "iso-8859-1"))
print(sniff_html(late))
print(sniff_html(b"<p>no declaration</p>"))
```

```
('utf-8', 'meta tag')
('iso-8859-1', 'HTTP header')
('utf-8', 'BOM')
('cp1252', 'default')
('cp1252', 'default')
```

Read the output line by line. A page with a meta tag is read as UTF-8; an HTTP header overrides the meta tag (priority order); a BOM beats the header; a meta tag that starts after 1100 bytes of comment is never seen, so the page falls back to the default and breaks; and a page with no declaration at all falls back too. So the practical advice is to send the HTTP charset, *and* put `<meta charset="utf-8">` as early as possible in the `<head>`, and save the file as UTF-8.

> Common mistake
> Fixing a symptom in one layer while another layer is still wrong, for example editing a page's meta tag while the server keeps sending a different charset in the header (which wins), or re-saving a file as

UTF-8 while the database column or HTTP header still says Latin-1. Trace the bytes through each boundary: file, server, header, page, browser.

Exercises

EXERCISE 1

Write `read_guess(path)` that reads a file's bytes and decodes them as UTF-8 (accepting a BOM, using `utf-8-sig`), falling back to Windows-1252 if that fails, and returns the text and which path it took. Test it on four files: plain UTF-8, UTF-8 with BOM, Windows-1252 with a euro sign, and ASCII only. What is the weakness of this approach?

View solution

EXERCISE 2

Write `normalise_newlines(text)` turning CR LF and lone CR into LF. Compare with `str.splitlines` on a string mixing all three endings. Then write the same two-line text with `newline` set to LF, CR LF and left at the default, and print the bytes of each. Which line depends on the machine?

View solution

EXERCISE 3

Write `decode_response(content_type, body)`: use the header's charset if present; otherwise try strict UTF-8, then fall back to Windows-1252. Try five cases, including a header charset that is wrong for the bytes. Which case fails, and what does that tell you about trusting headers?

View solution

CHAPTER 7 QUICK REFERENCE

- Text becomes bytes at every boundary; the reader needs the encoding from somewhere
- Python's default file encoding is platform-dependent (PEP 686 makes UTF-8 mode the default from 3.15); always pass `encoding=`

- `newline=` controls line ending translation: `"\n"`, `"\r\n"` or `""` (no translation on read); the default follows the platform
- HTTP: `Content-Type: text/html; charset=utf-8`; the value is case-insensitive, lowercase preferred
- JSON for interchange must be UTF-8, with no BOM; non-BMP characters may be written as escaped surrogate pairs
- HTML encoding priority: BOM, user override, HTTP charset, meta in the first 1024 bytes, other clues, locale default (windows-1252 for most locales)
- Put `<meta charset="utf-8">` early in the head, and send the header too
- `utf-8-sig` reads and writes the UTF-8 BOM; useful for some spreadsheet imports (check), wrong for JSON

NOT YET VERIFIED

Checked, through a summarising fetch tool and not by reading line by line, against: MDN's Content-Type page (the charset parameter and its example; it did not say what applies when charset is absent), the WHATWG HTML Standard section on determining the encoding (the priority list, the 1024-byte guidance, the locale-based defaults, the UTF-8 recommendation; the page was read only in part, the first 100,000 of about 457,000 characters), PEP 686 (UTF-8 mode default from 3.15, platform-dependent default, benefit mainly to Windows users) and RFC 8259 (UTF-8 requirement, no BOM, surrogate-pair escapes). Not source-checked, so from general knowledge or from this one machine: that spreadsheet programs often need a BOM to detect UTF-8, the csv module's advice on `newline=""`, the final status of the Python 3.15 change, the exact behaviour of the `email` package's header parsing on other Python versions, and the two machine-specific outputs (the first example and the default-newline line in Exercise 2). The page-encoding sniffer is a simplified teaching sketch, not the standard's algorithm. Sources: developer.mozilla.org (Content-Type), html.spec.whatwg.org, peps.python.org/pep-0686, rfc-editor.org/rfc/rfc8259.

COMING NEXT

Text Encoding and Unicode 8 looks at the database layer: collation, why one database's "utf8" is not UTF-8, and how column and index limits count characters.

Text Encoding and Unicode

Course 1 · Chapter 8 · Databases

A database is another boundary where text meets bytes, and it adds questions of its own. How is the text stored? What does "length" count? Is "É" equal to "é"? In what order do names sort? Is "Zoë" a duplicate of "Zoë"? Each database answers these through settings called the **character set** and the **collation**, and a wrong answer silently corrupts data or loses search results. This chapter uses SQLite (which ships with Python, so every example can be run) to show the behaviours, and the MySQL documentation to explain a famous trap.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 with its bundled SQLite, and the output shown is the real output. **No other database server was run**: statements about MySQL come from its documentation, and statements about other products are marked "(check)". The SQLite and MySQL facts come from their official documentation pages, each read through a summarising tool (see "Not Yet Verified" at the end). Details not checked against a source are marked "(check)".

How Text Is Stored and Counted

SQLite's documentation says a text value is "stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE)". Its `length()` function returns, for a string, "the number of Unicode codepoints (not bytes)" (up to the first NUL character), and for a blob the number of bytes. Casting to a blob therefore gives the byte count, and `length()` of the text matches Python's `len()`:

```
import sqlite3

con = sqlite3.connect(":memory:")
samples = ["e", "\u00e9", "e\u0301", "\u20ac", "\U0001F600", "\U0001F468\u200D\U0001F469
for s in samples:
    n_chars, n_bytes = con.execute("SELECT length(?), length(CAST(? AS BLOB))", (s, s)).
    print(f"{ascii(s):48} sql length={n_chars}  bytes={n_bytes}  python len={len(s)}")
```

```
'e'                                              sql length=1  bytes=1  python len=1
'\xe9'                                           sql length=1  bytes=2  python len=1
```

```
'e\u0301'                                          sql length=2  bytes=3  python len=2
'\u20ac'                                           sql length=1  bytes=3  python len=1
'\U0001f600'                                       sql length=1  bytes=4  python len=1
'\U0001f468\u200d\U0001f469\u200d\U0001f467'       sql length=5  bytes=18  python len=5
```

The database's "length" is the code point count from *Text Encoding and Unicode 1*: the decomposed é is 2, the family emoji is 5, and neither matches what a reader sees (*Text Encoding and Unicode 6*). Column definitions such as "up to 255 characters" in other products usually count code points too, but storage and index limits are often in bytes (check for your product). The difference matters when a character can take up to four bytes:

```
def fits_utf8mb3(s):
    return all(ord(c) <= 0xFFFF for c in s)

for s in ["caf\u00e9", "\u20ac100", "\u8a9e", "\U0001F600", "\U00010437"]:
    print(f"{ascii(s):16} fits 3-byte utf8: {fits_utf8mb3(s)!s:5} max bytes per char: {m

name = "\U0001F600" * 255
print(len(name), len(name.encode("utf-8")), len(("a" * 255).encode("utf-8")))
```

```
'caf\xe9'        fits 3-byte utf8: True  max bytes per char: 2
'\u20ac100'      fits 3-byte utf8: True  max bytes per char: 3
'\u8a9e'         fits 3-byte utf8: True  max bytes per char: 3
'\U0001f600'     fits 3-byte utf8: False max bytes per char: 4
'\U00010437'     fits 3-byte utf8: False max bytes per char: 4
255 1020 255
```

The last line is the arithmetic of a 255-character name: 255 bytes if all ASCII, 1,020 bytes if every character is outside the BMP. Space planning must allow for the worst case (check how your product measures index key length).

The "utf8" Trap in MySQL

The MySQL documentation describes two different Unicode character sets built on UTF-8. `utf8mb4` "requires a maximum of four bytes per multibyte character" and supports the BMP and supplementary characters. `utf8mb3` "supports only BMP characters and uses a maximum of three bytes per character"; for a supplementary character, "`utf8mb4` requires four bytes to store

it, whereas `utf8mb3` cannot store the character at all." The function above is exactly that test: an emoji or a rare script fails in a three-byte set. MySQL's name `utf8` is documented as a deprecated alias for `utf8mb3`, and the documentation says `utf8mb3` is deprecated, to be removed in a future major release, and that `utf8mb4` is recommended for all new applications. So a column declared as plain "utf8" in a MySQL database may not be UTF-8 in the full sense. Whether it errors, truncates or substitutes when you insert an emoji depends on the server's strict-mode settings (check; I have not run MySQL here). Moving from `utf8mb3` to `utf8mb4` is described as safe for existing data, since the old columns contain no supplementary characters.

Case, Equality and the ASCII-Only Default

SQLite is blunt about case. Its documentation says `lower()` "works for ASCII characters only" by default, that LIKE is case-insensitive only for ASCII (`'a' LIKE 'A'` is true, `'æ' LIKE 'Æ'` is false), and that the `NOCASE` collation folds only the 26 ASCII capitals. Non-ASCII case handling needs the ICU extension. These behaviours, confirmed here:

```
import sqlite3

con = sqlite3.connect(":memory:")
q = lambda sql: con.execute(sql).fetchone()[0]
print(q("SELECT 'a' LIKE 'A'"), q("SELECT '\u00e9' LIKE '\u00c9'"))
print(q("SELECT lower('\u00c9COLE ABC')") == "\u00c9cole abc")
print(q("SELECT '\u00e9' = '\u00c9' COLLATE NOCASE"), q("SELECT 'e' = 'E' COLLATE NOCASE
print(ascii("\u00c9COLE ABC".casefold()))
```

```
1 0
True
0 1
'\xe9cole abc'
```

Reading it: ASCII letters compare case-insensitively (1) but é and É do not (0); `lower()` lowered the ASCII letters and left the É upper case (the `True` line asserts that); the same ASCII-only rule applies to `NOCASE`; Python's `casefold()`, by contrast, handles all of it. Other products ship full Unicode collations (check which and how they are named). The general lesson is to find out, for your database, what its case-insensitive comparison really covers.

Equal to the Eye, Different to the Database

A database compares what it is given. The two spellings of "Zoë" from *Text Encoding and Unicode 5* are different values, so a uniqueness constraint accepts both, and a lookup by one does not find the other, unless the application normalizes on the way in:

```
import sqlite3
import unicodedata as ud

con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE users (name TEXT UNIQUE)")
composed, decomposed = "Zo\u00eb", "Zoe\u0308"
con.execute("INSERT INTO users VALUES (?)", (composed,))
con.execute("INSERT INTO users VALUES (?)", (decomposed,))
print(con.execute("SELECT count(*) FROM users").fetchone()[0])
print(con.execute("SELECT count(*) FROM users WHERE name = ?", (composed,)).fetchone()[0

con.execute("DELETE FROM users")
def add(name):
    try:
        con.execute("INSERT INTO users VALUES (?)", (ud.normalize("NFC", name),))
        return "added"
    except sqlite3.IntegrityError:
        return "duplicate"
print(add(composed), add(decomposed))
```

```
2
1
added duplicate
```

The table holds two "Zoë" rows, and a search for one finds only one. With NFC applied at the door the second insert is correctly rejected. Normalize where text *enters*, not only where it is compared.

Sorting: Binary Is Not Alphabetical

SQLite's default collation, `BINARY`, "compares string data using memcmp(), regardless of text encoding": it orders by bytes, not by what a dictionary does. That puts every capital before every

lowercase letter and every accented letter after "z". You can register your own collation, here a rough one based on the folding from Text Encoding and Unicode 5:

```
import sqlite3
import unicodedata as ud

words = ["zebra", "\u00c9clair", "apple", "\u00e9cole", "Zoo", "Eagle"]
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE w (word TEXT)")
con.executemany("INSERT INTO w VALUES (?)", [(w,) for w in words])
print([r[0] for r in con.execute("SELECT word FROM w ORDER BY word")])

def fold_key(s):
    return ud.normalize("NFD", s).casefold()

con.create_collation("folded", lambda a, b: (fold_key(a) > fold_key(b)) - (fold_key(a) <
print([r[0] for r in con.execute("SELECT word FROM w ORDER BY word COLLATE folded")])
```

```
['Eagle', 'Zoo', 'apple', 'zebra', 'Éclair', 'école']
['apple', 'Eagle', 'Éclair', 'école', 'zebra', 'Zoo']
```

The first line is the binary order; the second sorts "Éclair" and "école" with the E words. Even so, this key is crude: real languages differ (some sort accented letters as separate letters, and the right order for "ä" differs between languages, check). Serious applications use the database's own locale-aware collation or the ICU library rather than a hand-made key, and the collation also decides what `=` and `UNIQUE` mean.

Searching: Folded Shadow Columns

When you need "ecole" to find "École", a common pattern is to store a second, folded copy of the text for searching and match against that. This reuses the accent-stripping function from *Text Encoding and Unicode 5*:

```
import sqlite3
import unicodedata as ud

def fold(s):
    d = ud.normalize("NFD", s)
```

```
    return "".join(c for c in d if not ud.combining(c)).casefold()

con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE t (title TEXT, search TEXT)")
titles = ["\u00c9cole de Paris", "Ecole du soir", "Cr\u00e8me br\u00fbl\u00e9e", "Creme
con.executemany("INSERT INTO t VALUES (?, ?)", [(x, fold(x)) for x in titles])

def plain(term):
    return [r[0] for r in con.execute("SELECT title FROM t WHERE title LIKE ?", (f"%{ter

def folded(term):
    return [r[0] for r in con.execute("SELECT title FROM t WHERE search LIKE ?", (f"%{fo

for term in ["ecole", "\u00e9cole", "creme", "ZOE", "strasse"]:
    print(f"{term:8} plain={plain(term)}  folded={folded(term)}")
```

```
ecole    plain=['Ecole du soir']  folded=['École de Paris', 'Ecole du soir']
école    plain=[]  folded=['École de Paris', 'Ecole du soir']
creme    plain=['Creme fraiche']  folded=['Crème brûlée', 'Creme fraiche']
ZOE      plain=[]  folded=['Zoë']
strasse  plain=[]  folded=['straße']
```

The plain search misses most of what a user would expect: "école" finds nothing because the stored capital É is not folded to é, and "ZOE" misses "Zoë" because LIKE ignores case only for ASCII. The folded column finds them all, including "strasse" for "straße" (`casefold` expands ß to "ss"). The price: a second column to maintain, and you must apply the same `fold` to the search term. Whether folding accents is *right* depends on the language and the audience: in some languages an accent changes the word.

> Common mistake
> Treating the database as the place where Unicode "just works". It stores whatever bytes it is given in the encoding its settings dictate; it does not normalize, and its default case and sort rules may be ASCII-only. Check three settings for any database you use: the storage or column character set, the connection character set, and the collation. A mismatch in the connection setting is a classic cause of mojibake (see *Text Encoding and Unicode 7*).

Exercises

EXERCISE 1

For five strings (ASCII, accented, euros, emoji, CJK), use SQL to get the character count and the byte count, and test each against a limit of 5 characters and a limit of 5 bytes. Which strings pass one limit but fail the other, and why does it matter when a product measures a column in bytes?

View solution

EXERCISE 2

Make two tables: one with a `UNIQUE` name column using `COLLATE NOCASE`, one with a separate key column filled by NFKC plus `casefold()`. Insert "Zoe", "ZOE", "Zoë", "ZOË", the decomposed Zoë and a fullwidth-letter Zoë into both. Which inserts does each table reject, and which duplicates does only the key column catch?

View solution

EXERCISE 3

Build the folded search column from this chapter for six titles (including accented, German sharp-s and capital-accent cases). Compare a plain `LIKE` search with a folded search for five terms, and explain each place where the plain search fails.

View solution

CHAPTER 8 QUICK REFERENCE

- SQLite stores text as UTF-8 or UTF-16; `length()` counts code points for text and bytes for blobs; cast to a blob for the byte count
- MySQL: `utf8mb4` is up to 4 bytes and covers everything; `utf8mb3` (the deprecated "utf8" alias) is up to 3 bytes and cannot store characters above U+FFFF
- SQLite default `lower()`, LIKE and `NOCASE` fold ASCII only; the ICU extension adds full Unicode case folding
- The default `BINARY` collation compares bytes: capitals before lowercase, accented letters after z
- Databases do not normalize: normalize (NFC) on input, or the same text can be stored twice
- Use a folded shadow column (or a proper collation) for accent- and case-insensitive search; apply the same folding to the query

- Check character set, connection character set and collation for every database you use
- Limits may be in bytes even when lengths are in characters: plan for 4 bytes per character

NOT YET VERIFIED

Checked, through a summarising fetch tool and not by reading line by line, against the SQLite documentation pages on datatypes and collation, expressions (LIKE) and core functions (length, lower, instr, substr), and the MySQL reference manual pages on utf8mb4 and utf8mb3: the storage encodings, length semantics, ASCII-only case behaviour, the BINARY and NOCASE collations, the byte limits of the two MySQL sets, and the deprecation statements. The MySQL page that describes the "utf8" alias itself was not read, so the alias is stated only as the navigation titles of the pages I read show it. What MySQL does on inserting an unsupported character, how other products (and their index limits) handle collation and lengths, language-specific sort rules, and the accuracy of the crude folded sort key are from general knowledge or this one run (check). No MySQL, PostgreSQL or other server was run. Sources: sqlite.org/datatype3.html, sqlite.org/lang_expr.html, sqlite.org/lang_corefunc.html, dev.mysql.com/doc/refman/8.4 (charset-unicode-utf8mb4 and charset-unicode-utf8mb3).

COMING NEXT

Text Encoding and Unicode 9 turns to security: look-alike characters, invisible and direction-changing characters, and why validation must happen after normalization.

Text Encoding and Unicode

Course 1 · Chapter 9 · Security

Everything that makes Unicode useful also makes it a place to hide things. Characters from different alphabets can look identical. Some characters are invisible. Others change the direction in which text is displayed, so that what you read on screen is not the order in which a program reads it. This chapter shows the main tricks, with code to detect them, and the rule that ties the course together: **validate text only after you have normalized it**. The programs here are defensive checks, written to be run on suspicious text, not to produce it.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. Facts come from the Unicode Technical Standard #39 (security mechanisms), the Unicode Bidirectional Algorithm annex (UAX #9) and the Trojan Source project page, each read through a summarising tool (see "Not Yet Verified" at the end). One further older security report I tried to use is no longer maintained and held no usable content. Details not checked against a source are marked "(check)". The checks below are teaching examples, not a complete security solution.

Look-Alikes: Homoglyphs

Different scripts contain letters that look almost the same. The Cyrillic small letter a (`U+0430`) and the Latin small letter a (`U+0061`) are different code points, so a string using one is a different string. UTS #39 gives the classic example of "paypal" with the Cyrillic a, and calls such pairs *confusables*. Normalization does not help, because these characters are not canonically or compatibly equivalent: they are simply different letters that look alike.

```
import unicodedata as ud

pairs = [("Latin a", "a"), ("Cyrillic a", "\u0430"), ("Greek omicron", "\u03bf"), ("Lati
for label, ch in pairs:
    print(f"{label:14} U+{ord(ch):04X}  {ud.name(ch)}")

real = "paypal"
fake = "p\u0430ypal"
print(real == fake, real, fake)
```

```
print(len(real), len(fake), [f"{ord(c):04X}" for c in fake][:2])
print(real.isascii(), fake.isascii())
```

```
Latin a        U+0061  LATIN SMALL LETTER A
Cyrillic a     U+0430  CYRILLIC SMALL LETTER A
Greek omicron  U+03BF  GREEK SMALL LETTER OMICRON
Latin o        U+006F  LATIN SMALL LETTER O
Cyrillic o     U+043E  CYRILLIC SMALL LETTER O
False paypal paypal
6 6 ['0070', '0430']
True False
```

unequal and the second is not ASCII. Both have six characters, so a length check sees nothing. Two defences are common. The simplest is to restrict identifiers such as user names to ASCII, or to a single script. UTS #39 describes graduated *restriction levels* (from ASCII-only and single-script up to unrestricted), and a detection method: reduce each string to a *skeleton* by mapping every character to a prototype, and treat strings with equal skeletons as confusable. The standard stresses that skeletons are for internal comparison and should not be shown or stored in place of the original. A small mixed-script check, a rough version of the first idea, uses the Unicode character name to find the script:

```
import unicodedata as ud

def scripts(s):
    found = set()
    for c in s:
        if c.isalpha():
            found.add(ud.name(c).split()[0])
    return sorted(found)

for word in ["paypal", "p\u0430ypal", "\u043f\u0440\u0438\u0432\u0435\u0442", "caf\u00e9
    sc = scripts(word)
    print(f"{ascii(word):22} scripts={sc}  mixed={len(sc) > 1}")
```

```
'paypal'               scripts=['LATIN']  mixed=False
'p\u0430ypal'          scripts=['CYRILLIC', 'LATIN']  mixed=True
'\u043f\u0440\u0438\u0432\u0435\u0442' scripts=['CYRILLIC']  mixed=False
'caf\xe9'              scripts=['LATIN']  mixed=False
```

The mixed-script word is caught; an all-Cyrillic word is not flagged, because it is a legitimate word in one script (a fully Cyrillic string made only of look-alike letters would still pass: another reason to prefer a real library implementing UTS #39 for anything important, check what is available for your language). My first-word-of-the-name trick is a crude stand-in: UTS #39 uses the Unicode Script_Extensions property and treats Common and Inherited characters as matching any script, which this sketch ignores.

Invisible Characters

Unicode has characters that take no space on screen. Python classifies them as category `Cf` (format). Some are essential in some languages (the zero-width joiner of *Text Encoding and Unicode 6* builds emoji sequences), but in a user name or a file name they are a way to make two different strings look the same:

```
import unicodedata as ud

invisible = ["\u200b", "\u200c", "\u200d", "\u2060", "\ufeff", "\u00ad"]
for c in invisible:
    print(f"U+{ord(c):04X}  {ud.name(c):24} category={ud.category(c)}  isprintable={c.isp

name1 = "admin"
name2 = "admin\u200b"
print(name1 == name2, len(name1), len(name2))
print(ascii(name2))
```

```
U+200B  ZERO WIDTH SPACE         category=Cf  isprintable=False
U+200C  ZERO WIDTH NON-JOINER    category=Cf  isprintable=False
U+200D  ZERO WIDTH JOINER        category=Cf  isprintable=False
U+2060  WORD JOINER              category=Cf  isprintable=False
U+FEFF  ZERO WIDTH NO-BREAK SPACE category=Cf  isprintable=False
U+00AD  SOFT HYPHEN              category=Cf  isprintable=False
False 5 6
'admin\u200b'
```

"admin" with a trailing zero-width space prints as "admin" but is a different, longer string. You have already met two of these characters: `U+FEFF`, the byte order mark (*Text Encoding and Unicode 4*), and the joiner. UTS #39 removes Default_Ignorable characters when building

skeletons for exactly this reason. Always display suspicious strings through `ascii()` or a similar escaping view when debugging.

Direction Changers: Bidirectional Controls

Scripts such as Arabic and Hebrew run right to left. The Unicode Bidirectional Algorithm (UAX #9) therefore defines invisible control characters that set the direction of a stretch of text. Per UAX #9 these are: embeddings (LRE `U+202A`, RLE `U+202B`), overrides (LRO `U+202D`, RLO `U+202E`), the terminator PDF (`U+202C`), and the isolates LRI `U+2066`, RLI `U+2067`, FSI `U+2068` with their terminator PDI `U+2069`. The standard itself advises avoiding overrides where possible because of security concerns.

The security problem, named **Trojan Source** by researchers Nicholas Boucher and Ross Anderson (University of Cambridge; USENIX Security 2023), is that compilers and interpreters "adhere to the logical ordering of source code, not the visual order", so bidirectional controls hidden in comments or strings can make code look different in a review from what runs. Their page lists CVE-2021-42574 for the bidirectional attack and CVE-2021-42694 for a related homoglyph attack, and recommends that tools warn or error on unterminated bidi controls and on confusable identifiers, and that editors and repository front ends make such characters visible. A detector is short:

```
import unicodedata as ud

BIDI = {0x202A: "LRE", 0x202B: "RLE", 0x202C: "PDF", 0x202D: "LRO", 0x202E: "RLO",
        0x2066: "LRI", 0x2067: "RLI", 0x2068: "FSI", 0x2069: "PDI"}
for cp, abbr in BIDI.items():
    print(f"U+{cp:04X} {abbr}  {ud.name(chr(cp))}  bidi class={ud.bidirectional(chr(cp))}

def bidi_controls(s):
    return [(i, BIDI[ord(c)]) for i, c in enumerate(s) if ord(c) in BIDI]

line = "access = 'user\u202e\u2066 // check admin\u2069\u2066'"
print(bidi_controls(line))
print(ascii(line))
```

```
U+202A LRE  LEFT-TO-RIGHT EMBEDDING  bidi class=LRE
U+202B RLE  RIGHT-TO-LEFT EMBEDDING  bidi class=RLE
U+202C PDF  POP DIRECTIONAL FORMATTING  bidi class=PDF
```

```
U+202D LRO  LEFT-TO-RIGHT OVERRIDE  bidi class=LRO
U+202E RLO  RIGHT-TO-LEFT OVERRIDE  bidi class=RLO
U+2066 LRI  LEFT-TO-RIGHT ISOLATE  bidi class=LRI
U+2067 RLI  RIGHT-TO-LEFT ISOLATE  bidi class=RLI
U+2068 FSI  FIRST STRONG ISOLATE  bidi class=FSI
U+2069 PDI  POP DIRECTIONAL ISOLATE  bidi class=PDI
[(14, 'RLO'), (15, 'LRI'), (31, 'PDI'), (32, 'LRI')]
"access = 'user\u202e\u2066 // check admin\u2069\u2066'"
```

The names and positions are the printed proof. Without the detector, the line would simply display with its pieces reordered. Legitimate right-to-left text uses these controls too, so a detector should flag them for human review in source code, file names and identifiers, not delete them everywhere.

Order of Operations: Normalize First, Then Validate

Several earlier parts of this course have pointed at one rule. If you check text for forbidden content and *then* normalize it, the normalization can create what the check was meant to prevent. Compatibility normalization turns fullwidth angle brackets into ordinary ones, so a filter that looks for ASCII markup misses the fullwidth version, and the later step produces the dangerous text:

```
import unicodedata as ud

def blocked(s):
    return "<script" in s.lower()

attack = "\uff1cscript\uff1e"          # fullwidth < and >
print(blocked(attack))
after = ud.normalize("NFKC", attack)
print(ascii(after), blocked(after))

def safe_check(s):
    return blocked(ud.normalize("NFKC", s))
print(safe_check(attack))
```

```
False
'<script>' True
True
```

The first `False` is the bug: the filter passed text that becomes `<script>` after NFKC. Checking the normalized form (last line) catches it. The same logic applies to the overlong UTF-8 forms of *Text Encoding and Unicode 2*: decode strictly first, then check the decoded text, never the raw bytes. (A blocklist is a weak defence on its own; the point here is only the ordering. Better defences allow known-good input and escape output for its context, check.) Combining all the checks of this chapter into one reporting function:

```
import unicodedata as ud

BIDI = set(range(0x202A, 0x202F)) | set(range(0x2066, 0x206A))

def problems(s):
    found = []
    n = ud.normalize("NFC", s)
    if n != s:
        found.append("not NFC")
    if any(ud.category(c) == "Cf" and ord(c) not in (0x200D, 0x200C) for c in s):
        found.append("invisible format character")
    if any(ord(c) in BIDI for c in s):
        found.append("bidi control")
    scripts = {ud.name(c).split()[0] for c in s if c.isalpha()}
    if len(scripts) > 1:
        found.append("mixed scripts: " + "+".join(sorted(scripts)))
    if any(ud.category(c).startswith("C") and c not in "\t\n\r" and ud.category(c) != "C
        found.append("control or unassigned")
    return found

tests = ["alice", "p\u0430ypal", "admin\u200b", "Zoe\u0308", "user\u202e", "\u043f\0440
for t in tests:
    print(f"{ascii(t):28} {problems(t) or 'ok'}")
```

```
'alice'                      ok
'p\u0430ypal'                ['mixed scripts: CYRILLIC+LATIN']
'admin\u200b'                ['invisible format character']
'Zoe\u0308'                  ['not NFC']
'user\u202e'                 ['invisible format character', 'bidi control']
'\u043f\0440\0438\0432\0435\0442' ok
'bad\x07bell'                ['control or unassigned']
```

Note the policy choices inside the function: it tolerates the joiner and non-joiner (needed by some scripts and by emoji, check which your users need), accepts a fully Cyrillic word, and

reports rather than rejects. A real policy depends on what the field is: a password field should accept almost anything, a user name should accept very little, and a file name sits somewhere between.

> Common mistake
> Checking raw input and then transforming it, or showing the original string in a log and assuming it is safe to read. Decode strictly, normalize, validate, and only then use the text. Log suspicious strings in escaped form so that invisible and direction-changing characters cannot hide in your own logs.

Exercises

EXERCISE 1

Write `scripts(s)` returning the scripts used in a string (the first word of each letter's Unicode name) and classify six names as ASCII-only, single script or MIXED. Include a name using a Greek omicron, one using two Cyrillic o's, an all-Cyrillic name and an accented Latin name. What does the all-Cyrillic result show about the limits of the check?

View solution

EXERCISE 2

A site reserves the names "admin" and "root". Write `key(name)` using NFKC, `casefold()` and removal of format characters (category Cf), and test eight variants: upper case, fullwidth letters, a trailing zero-width space, a soft hyphen in the middle, and a Cyrillic o. Which variants does the naive check miss, and which does even your key miss?

View solution

EXERCISE 3

Write `scan(text)` that reports the line, column and name of every bidirectional control in a multi-line text, and `strip_bidi(text)` that removes them. Run both on a short sample containing a comment with two such characters, and compare the escaped view of the line before and after.

View solution

CHAPTER 9 QUICK REFERENCE

- Homoglyphs (Cyrillic a vs Latin a) are different letters, not equivalents: normalization does not merge them; restrict scripts or use UTS #39 skeletons
- Invisible format characters (category Cf: zero-width space, joiners, soft hyphen, BOM) make different strings look identical
- Bidi controls LRE, RLE, LRO, RLO, PDF (U+202A-202E) and LRI, RLI, FSI, PDI (U+2066-2069) can make code display differently from how it runs (Trojan Source)
- Decode strictly, normalize, then validate; checking before normalizing lets fullwidth or compatibility forms slip past
- Log suspicious text with `ascii()` so hidden characters become visible
- Detectors should flag for review rather than delete: legitimate text uses joiners and bidi controls
- Prefer allow-lists and context-specific output escaping to blocklists (check against current guidance)

NOT YET VERIFIED

Checked, through a summarising fetch tool and not by reading line by line, against: UTS #39 (confusables and skeletons, mixed-script detection, restriction levels; the page was read except its final few thousand characters), the Trojan Source project page (the attack idea, the quoted sentence, the two CVE identifiers, the authors and venue, the recommended defences) and UAX #9 (the code points and names of the bidirectional controls and the advice about overrides; read only in part, the first 100,000 of about 134,000 characters). The older Unicode technical report on security considerations is marked as no longer maintained and its page held none of the content I needed, so nothing is drawn from it. From general knowledge or from this run, and so flagged (check): how fonts render the look-alike letters, which libraries implement the UTS #39 mechanisms for a given language, the need for joiners in specific scripts, and the advice to prefer allow-lists. The checkers in this chapter are teaching sketches, not complete defences. Sources: unicode.org/reports/tr39/, trojansource.codes, unicode.org/reports/tr9/.

COMING NEXT

Text Encoding and Unicode 10 brings the whole course together: a diagnosis workflow for any encoding problem, the tools to use, and a capstone that tracks a garbled string back to its cause.

Text Encoding and Unicode

Course 1 · Chapter 10 · Debugging Encodings

You now know what text is (*Text Encoding and Unicode 1*), how UTF-8 and its rivals are built (*Text Encoding and Unicode 2*, *3* and *4*), why equal-looking strings differ (*Text Encoding and Unicode 5* and *6*), and where encodings are declared or lost (*Text Encoding and Unicode 7* and *8*), and the security angle (*Text Encoding and Unicode 9*). This final chapter turns that knowledge into a repeatable debugging method, builds a small toolkit, and then solves a realistic damaged string end to end.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every program in this chapter was run on Python 3.14 and the output shown is the real output. This chapter introduces no new reference material: the facts it uses were established, with sources, in the earlier parts of the course, and the new material (the workflow and the toolkit) is my own design, so it is a worked method and not a standard. Details not checked against a source are marked "(check)". The toolkit uses heuristics and can be wrong on unusual text.

The Six-Step Method

1. **Look at the bytes and code points, not the rendered text.** Print `ascii(s)` and the UTF-8 hex. A screen hides invisible characters and may itself be using the wrong font or encoding.
2. **Recognise the pattern.** Most damage has a recognisable shape (table below).
3. **Decide: reversible or lost?** If the original bytes are still all present (just misread) you can reverse it. If a step replaced characters with `?` or `U+FFFD`, the information is gone and you must re-fetch the source.
4. **Find the boundary.** Text passes through boundaries (file, network, database, program). Test the text after each one and find the first where it stops matching.
5. **Fix at the source.** Declare or set the right encoding at the faulty boundary. Do not scatter repairs through the program.
6. **Prevent.** Always pass an encoding, normalize on input, and keep test data with non-ASCII characters, emoji and a decomposed accent in it.

Step 1: Look at the Real Data

```
def show(s):
    print("text :", s)
    print("repr :", ascii(s))
    print("utf-8:", s.encode("utf-8", errors="surrogatepass").hex(" "))

show("caf\u00e9 \u2014 \u201cok\u201d")
```

```
text : café — “ok”
repr : 'caf\xe9 \u2014 \u201cok\u201d'
utf-8: 63 61 66 c3 a9 20 e2 80 94 20 e2 80 9c 6f 6b e2 80 9d
```

The `repr` line shows exactly which code points are present (including any invisible ones, *Text Encoding and Unicode 9*), and the hex line shows the bytes, which is what you compare with what you expect from a file or a network capture.

Step 2: Recognise the Pattern

Take one clean string and damage it in the common ways, to learn what each looks like:

```
original = "caf\u00e9 \u2014 \u201cok\u201d"
raw = original.encode("utf-8")
for wrong in ["cp1252", "cp437"]:
    print(f"{wrong:8} {raw.decode(wrong, errors='replace')}")
print("ascii    ", raw.decode("ascii", errors="replace"))
print("lossy    ", original.encode("ascii", errors="replace").decode("ascii"))
```

```
cp1252   cafÃ© â€” â€œokâ€
cp437    caf├⌐ ΓÇö ΓÇ£okΓÇ¥
ascii     caf   ok
lossy     caf? ? ?ok?
```

The Latin-1 version of this damage looks much the same but with invisible control characters in place of the Windows-1252 punctuation. Use this as a field guide:

YOU SEE	LIKELY CAUSE	REVERSIBLE?
Ã© Ã¨ Ã¶ and â€™ â€œ	UTF-8 bytes read as Windows-1252 or Latin-1 (*Text Encoding and Unicode 3*)	Usually yes, if no byte was replaced; the last quote above ended in U+FFFD (the replacement character), because byte 9D is undefined in Windows-1252, so that one is lost
Ã¯Â»Â¿ at the start	A UTF-8 byte order mark read as text, often after two layers (*Text Encoding and Unicode 4*)	Yes: decode with `utf-8-sig`
Ã followed by more Ã, Â and ,	Several layers of the same mistake	Yes, one layer at a time
 (U+FFFD)	A decoder replaced bytes it could not read	No: re-read the original source
? where a letter should be	An encoder replaced characters it could not represent	No
é or `\u00e9` shown literally	Escaped text that was never unescaped (*Text Encoding and Unicode 7*)	Yes: unescape at the right layer
Letters separated by invisible NULs	16-bit text read as 8-bit (*Text Encoding and Unicode 4*)	Yes: decode as UTF-16
Empty boxes only	Characters are right but the font lacks glyphs	Nothing to repair: change font

Steps 3 to 5: A Toolkit and a Capstone

For the reversible case, "UTF-8 read as Windows-1252", the repair is to encode the garbled text back to bytes with the wrong codec, then decode those bytes as UTF-8. Real data complicates this in two ways: some bytes are undefined in Windows-1252 (so many tools turn them into the matching control character, which we copy), and damage may have happened more than once. The program below also contains a guard: it only accepts a repair if it reduces the number of tell-tale marker sequences, so clean text, and legitimate text such as the Portuguese city name in capitals, is left alone.

```
MARKERS = ("\u00c3", "\u00c2", "\u00e2\u20ac", "\u00ef\u00bb\u00bf")

def suspicion(s):
    return sum(s.count(m) for m in MARKERS)

def lenient_decode(raw):
    """Windows-1252, but the five undefined bytes become the matching control characters
    out = []
    for b in raw:
        try:
            out.append(bytes([b]).decode("cp1252"))
        except UnicodeDecodeError:
            out.append(chr(b))
    return "".join(out)

def lenient_encode(s):
    out = bytearray()
    for c in s:
        try:
            out += c.encode("cp1252")
        except UnicodeEncodeError:
            if ord(c) > 0xFF:
                raise
            out.append(ord(c))
    return bytes(out)

def undo_once(s):
    """Try to reverse one layer of 'UTF-8 bytes read as Windows-1252'."""
    try:
        fixed = lenient_encode(s).decode("utf-8")
    except (UnicodeEncodeError, UnicodeDecodeError):
        return None
    return fixed if suspicion(fixed) < suspicion(s) else None

def peel(s, max_layers=5):
    layers = 0
    while layers < max_layers:
        fixed = undo_once(s)
        if fixed is None:
            break
        s = fixed
        layers += 1
    return s, layers
```

```
once = lenient_decode("caf\u00e9 \u2014 \u201cok".encode("utf-8"))
print(once)
print(peel(once))
for fine in ["already fine \u00e9", "S\u00c3O PAULO"]:
    print(peel(fine))

# capstone: a BOM, a curly apostrophe, accents, a euro sign, a dash and a quote, damaged
original = "\ufeffZo\u00eb\u2019s caf\u00e9 costs 5 \u20ac \u2014 \u201copen"
layer1 = lenient_decode(original.encode("utf-8"))
layer2 = lenient_decode(layer1.encode("utf-8"))
print(ascii(original))
print(layer1)
print(layer2.replace(chr(0x9d), "<9D>"))
fixed, layers = peel(layer2)
print(layers, fixed == original)
print(ascii(fixed[:6]))
print(fixed.lstrip("\ufeff"))
```

```
cafÃ© â€” â€œok
('café — “ok', 1)
('already fine é', 0)
('SÃO PAULO', 0)
'\ufeffZo\xeb\u2019s caf\xe9 costs 5 \u20ac \u2014 \u201copen'
ï»¿ZoÃ«â€™s cafÃ© costs 5 â‚¬ â€” â€œopen
Ã¯Â»Â¿ZoÃƒÂ«Ã¢â‚¬â„¢s cafÃƒÂ© costs 5 Ã¢â€šÂ¬ Ã¢â‚¬<9D> Ã¢â‚¬Å“open
2 True
'\ufeffZo\xeb\u2019s'
Zoë’s café costs 5 € — “open
```

The capstone follows the method. Step 1: the repr of the original shows an invisible `U+FEFF` (a BOM) at the front. Step 2: the first damaged copy shows "ï»¿" and "Ã«", the second shows the same marks twice over ("Ã¯Â»Â¿"): two layers. Step 3: every byte survived, so the damage is reversible. Step 4: `peel` reports `2` layers and recovers the original exactly (`True`), which means the fault happened twice, that is, at two boundaries. Step 5 would be to find those two boundaries and set the encoding correctly there; step 6, to add a test containing a BOM, an accent and an euro sign. The final line strips the BOM, which should be removed once its job is done. As a check on the guard, "SÃO PAULO" was left untouched because reversing it would not produce valid UTF-8.

Finding the Boundary

Step 4 is a search: run the text through the stages in order and report the first stage whose output differs from the original. Here each stage is a real round trip (file, JSON, database, export), and in each run we inject a fault at one stage:

```
import json, sqlite3, tempfile, pathlib

original = "Zo\u00eb\u2019s caf\u00e9 \u20ac5"
tmp = pathlib.Path(tempfile.mkdtemp())

def lenient_decode(raw):
    out = []
    for b in raw:
        try:
            out.append(bytes([b]).decode("cp1252"))
        except UnicodeDecodeError:
            out.append(chr(b))
    return "".join(out)

def file_ok(s):
    p = tmp / "a.txt"
    p.write_text(s, encoding="utf-8", newline="\n")
    return p.read_text(encoding="utf-8")

def json_ok(s):
    return json.loads(json.dumps({"v": s}))["v"]

def db_ok(s):
    con = sqlite3.connect(":memory:")
    con.execute("CREATE TABLE t (v TEXT)")
    con.execute("INSERT INTO t VALUES (?)", (s,))
    return con.execute("SELECT v FROM t").fetchone()[0]

def wrong_read(s):
    return lenient_decode(s.encode("utf-8"))

def to_ascii(s):
    return s.encode("ascii", errors="replace").decode("ascii")

def first_bad_stage(text, stages):
    current = text
    for name, fn in stages:
        current = fn(current)
        if current != text:
            return name, current
```

```
    return None, current

names = ["file", "json", "database", "export"]
good = {"file": file_ok, "json": json_ok, "database": db_ok, "export": json_ok}
for fault, bad in [(None, None), ("file", wrong_read), ("database", wrong_read), ("expor
    stages = [(n, bad if n == fault else good[n]) for n in names]
    name, result = first_bad_stage(original, stages)
    print(f"fault in {fault!s:9} -> first bad stage: {name!s:9} {result}")
```

```
fault in None      -> first bad stage: None      Zoë’s café €5
fault in file      -> first bad stage: file      ZoÃ«â€™s cafÃ© â‚¬5
fault in database  -> first bad stage: database  ZoÃ«â€™s cafÃ© â‚¬5
fault in export    -> first bad stage: export    Zo??s caf? ?5
```

Each injected fault is found at its own stage. Without a fault, no stage is reported. The last row is the irreversible case, with question marks: it can be found, but not repaired. The method scales: in a real system the "stages" are captures and log lines taken at each boundary, and the comparison is done by eye against the hex dump from step 1.

> Common mistake
> Repairing garbled text with a fix-up function at the point where you noticed it. That hides the real fault, which will return with the next record. Use a repair function like `peel` only to rescue existing damaged data, then fix the boundary that caused it. Also never repair blindly: keep a copy of the original, and review a sample before bulk-fixing.

What the Course Gave You

PART	THE IDEA TO KEEP
Text Encoding and Unicode 1	Bytes, code points and what a reader sees are three different counts
Text Encoding and Unicode 2	UTF-8 is a small, strict bit pattern; decoders must refuse overlong forms and surrogates
Text Encoding and Unicode 3	Legacy encodings assign the upper half of a byte differently; "no error" proves nothing

PART	THE IDEA TO KEEP
Text Encoding and Unicode 4	UTF-16 uses surrogate pairs; byte order marks tell the order but are not always present
Text Encoding and Unicode 5	Normalize (NFC) before comparing; use NFKC only for matching
Text Encoding and Unicode 6	Count, cut and reverse by grapheme cluster when the audience is a person
Text Encoding and Unicode 7	Say the encoding at every boundary: files, headers, pages
Text Encoding and Unicode 8	Databases store what they are given; check character set, connection and collation
Text Encoding and Unicode 9	Decode strictly, normalize, then validate; make invisible characters visible

Exercises

EXERCISE 1

Write `report(s)` that checks a string for the symptoms in the field guide (replacement characters, a question mark between letters, BOM bytes shown as text, mojibake markers, numeric entities, literal backslash-u escapes, NUL characters) and returns a list of findings. Try it on eight samples, including a clean one.

View solution

EXERCISE 2

Using the toolkit from this chapter, damage a clean string by zero, one, two and three layers of "UTF-8 read as Windows-1252" and check that `peel` recovers the original each time. Then show two irreversible cases (ASCII replacement with question marks, and UTF-8 decoded as ASCII with replacement) and what `peel` does with them.

View solution

EXERCISE 3

Extend the boundary search: build a four-stage pipeline (file, JSON, database, export) with correct round-trip stages, then inject a fault at each stage in turn (a wrong-encoding read at the file, the database, and a lossy ASCII export) and confirm `first_bad_stage` names the right stage each time. What does the output of the lossy case tell you about repair?

View solution

CHAPTER 10 QUICK REFERENCE

- Method: look at bytes, recognise the pattern, decide reversible or lost, find the boundary, fix at the source, prevent
- `ascii(s)` and `s.encode("utf-8").hex(" ")` show what is really there
- Ã©, â€™, Ã¯Â»Â¿ mean UTF-8 read as Windows-1252/Latin-1; U+FFFD and ? mean the data is already lost
- Reverse one layer: `s.encode(wrong).decode("utf-8")`, copying the tolerant handling of the undefined bytes; guard against false repairs
- A faulty boundary is found by comparing the text after each stage with the original
- Repair functions rescue existing data; the real fix is to declare the right encoding where the fault happens
- Test with non-ASCII data: accents, emoji, a decomposed accent, a BOM

NOT YET VERIFIED

This chapter's facts are the ones established with sources in the earlier parts of the course (the Windows-1252 undefined bytes, the BOM bytes, the behaviour of the replacement handlers); it cites no new external source. The field guide, the six-step method, the marker list and the guard heuristic are my own design from this course, not an established standard (check against your own data). The claim that many tools map the five undefined bytes to control characters is from the Windows-1252 discussion in *Text Encoding and Unicode 3* (which cited one source for the existence of such mappings), and that the lenient decoder reproduces any particular tool's behaviour is not verified (check). The Portuguese city-name example shows a false positive being avoided, not a general guarantee. The font remark in the table (boxes mean missing glyphs) is general knowledge (check).

COURSE COMPLETE

That is the end of *Text Encoding and Unicode*. You can now say what a string is made of at three levels, read and write the main encodings by hand, compare and search text safely, send it across boundaries without damage, defend against the main tricks, and track a garbled string back to its cause. Keep the field guide table and the six steps; they will outlast any particular library.