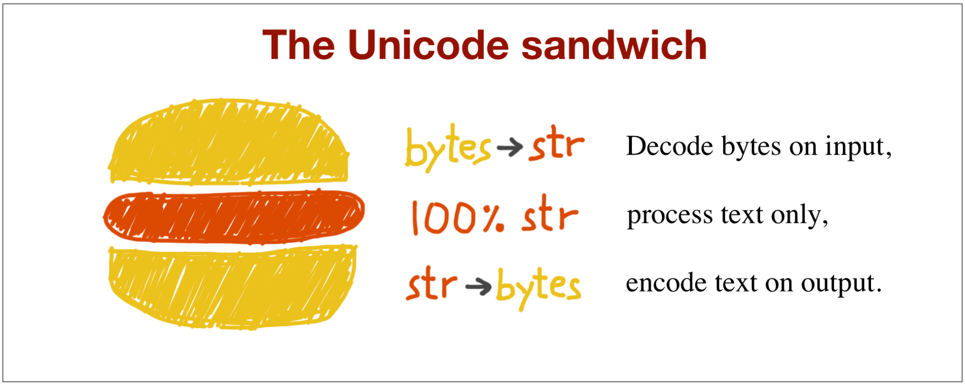

Decode bytes on input,

process text only,

encode text on output.

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

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

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

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

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

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

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

[8] I first saw the term "Unicode sandwich" in Ned Batchelder's excellent *Pragmatic Unicode* talk at US PyCon 2012

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